VAX-11
Symbolic Debugger Reference Manual

Order No. AA-D026D-TE

May 1982

This manual explains and illustrates the features of the VAX-11 Symbolic Debugger for both high-level and assembly language programmers.

REVISION/UPDATE INFORMATION: This revised document supersedes the VAX-11 Symbolic Debugger Reference Manual (Order No. AA-D026C-TE).

SOFTWARE VERSION: VAX/VMS Version 3.0

digital equipment corporation · maynard, massachusetts

First Printing, August 1978
Revised, March 1980
Revised, May 1981
Revised, May 1982

Printed in U.S.A.

The postpaid READER'S COMMENTS form on the last page of this document requests the user's critical evaluation to assist in preparing future documentation.

ZK2134

HOW TO ORDER ADDITIONAL DOCUMENTATION

In Continental USA and Puerto Rico call 800-258-1710

In New Hampshire, Alaska, and Hawaii call 603-884-6660

In Canada call 613-234-7726 (Ottawa-Hull)
800-267-6146 (all other Canadian)

DIRECT MAIL ORDERS (USA & PUERTO RICO)*

Digital Equipment Corporation
P.O. Box CS2008
Nashua. New Hampshire 03061

*Any prepaid order from Puerto Rico must be placed with the local Digital subsidiary (809-754-7575)

DIRECT MAIL ORDERS (CANADA)

Digital Equipment of Canada Ltd.
940 Belfast Road
Ottawa, Ontario K1G 4C2
Attn: A&SG Business Manager

DIRECT MAIL ORDERS (INTERNATIONAL)

Digital Equipment Corporation
A&SG Business Manager
c/o Digital's local subsidiary or approved distributor

Internal orders should be placed through the Software Distribution Center (SDC). Digital Equipment Corporation, Northboro, Massachusetts 01532

2/83-14

CONTENTS

CONTENTS

CONTENTS

Page

FIGURES

TABLES

PREFACE

MANUAL OBJECTIVES

This manual explains how to use the VAX-11 Symbolic Debugger. It is a reference manual that discusses debugger features and capabilities in language-independent terminology; it is not a source of language-specific information.

INTENDED AUDIENCE

Programmers at all levels of experience can use this manual effectively.

Familiarity with the VAX-11 architecture and with the VAX/VMS operating system is helpful but not essential to the effective use of this manual.

STRUCTURE OF THIS DOCUMENT

This manual contains 9 chapters.

- Chapter 1 provides a short description of debugger features, discusses debugger command format, and explains how to begin, end, and interrupt a debugging session.
- Chapter 2 discusses the use of symbols in debugging, the different kinds of symbols, the scope of symbols in the source language, and the debugger's interpretation of symbols.
- Chapter 3 discusses the referencing of program locations by means of simple addresses and address expressions and explains how type is associated with program locations.
- Chapter 4 explains how to examine, deposit, and evaluate program data, and how to use mode settings to influence the entry and display of program information.
- Chapter 5 explains how to start, suspend, and monitor program execution.
- Chapter 6 discusses log files and command procedures.
- Chapter 7 discusses assembly-level debugging.
- Chapter 8 describes how to display source language statements.
- Chapter 9 provides an alphabetical listing and description of debugger commands with information about command format, qualifiers, and parameters.

ASSOCIATED DOCUMENTS

Information on debugging in a particular language may be found in the documentation furnished with that language, usually in the appropriate language user's guide.

Information on the linking of programs and on shareable images may be found in the VAX-11 Linker Reference Manual.

Information on the VAX-11 architecture may be found in the VAX Architecture Handbook.

Information on various components of the VAX/VMS operating system may be found in the VAX-11 documentation set. See the VAX-11 Information Directory and Index for a full listing of available VAX/VMS documentation.

CONVENTIONS USED IN THIS DOCUMENT

Convention	Meaning
Uppercase words and letters	Uppercase words and letters, used in examples, indicate that you should type the word or letter exactly as shown.
Lowercase words and letters	Lowercase words and letters, used in format examples, indicate that you are to substitute a word or value of your choice.
quotation marks apostrophes	The term "quotation marks" refers to double quotation marks ("). The term "apostrophe" refers to a single quotation mark (').
[ ]	Square brackets indicate that the enclosed item is optional (except in file specifications where square brackets delimit directory names).
...	A horizontal ellipsis indicates that the preceding item(s) can be repeated one or more times. For example: file-spec[,file-spec...]
. . .	A vertical ellipsis indicates that not all of the statements in an example or figure are shown.
$ RUN $_File:	In examples of commands you enter and system responses, all output lines and prompting characters that the system prints or displays are shown in black letters. All the lines you type are shown in red letters.
RET	A symbol with a 1- to 3-character abbreviation indicates a key that you press on the terminal.

CTRL/X — The symbol CTRL/x indicates a control key sequence, which you enter by pressing the key labeled CTRL while simultaneously pressing another key, for example, <CTRL/C>, <CTRL/Y>, <CTRL/O>. In examples, this control key sequence is shown as ^x, for example, ^C, ^Y, ^O, because that is how the system echoes control key sequences.

Unless otherwise noted, all numeric values are represented in decimal notation.

Unless otherwise specified, you terminate commands by pressing the RETURN key.

SUMMARY OF TECHNICAL CHANGES

This manual describes Version 3.0 of the VAX-11 Symbolic Debugger. The following are technical changes that are new with this release:

The debugger now supports Version 2.0 of VAX-11 COBOL and Version 1.2 of VAX-11 PL/I.

There are six new debugger types that you may use in the entry or display of data: FLOAT, D_FLOAT, G_FLOAT, H_FLOAT, QUADWORD, and OCTAWORD.

In those languages that support the feature, the debugger now allows the display of lines of source code during a debugging session. Chapter 8 describes source line display.

When an exception breakpoint is activated, the issuing of a GO command without an address expression parameter resignals the exception and thereby allows any user-declared exception handler to gain control; previously, control bypassed any user-declared exception handler. Further, the issuing of a STEP command directly following an exception breakpoint hit is now illegal.

The debugger now prefixes the file name of each shareable image included in the executable image with the name SHARE$. You can use this new name in debugging shareable images.

CHAPTER 1

GETTING STARTED

This chapter introduces debugging; outlines debugger features; describes debugger command format; and discusses how to start, interrupt, and end a debugging session.

1.1 INTRODUCTION TO DEBUGGING

The normal behavior of a program is to transform a set of inputs into a set of outputs according to the logical procedures in the program. Abnormal behavior such as incorrect output, premature program termination, or infinite looping indicates that one or more programming errors (bugs) exist.

Debugging is the process of locating programming errors in a successfully compiled program that behaves abnormally when run. The VAX-11 Symbolic Debugger (or, simply, the debugger) is a special program to help you locate these programming errors. After you locate the errors, you make corrections in your source program; then you compile (or assemble), link, and execute the corrected version.

After checking your program for typographical or syntactical errors that might not have been detected by the compiler, you should formulate a debugging plan based on the kind of abnormal behavior displayed by the program.

Abnormal termination of the program suggests either a logical error or an error caused by something in the environment. Since abnormal termination occurs reasonably soon after the error that caused it, a useful approach to locating the error is to backtrack from the point of termination. Using the debugger, you can examine the output of the program at points just before termination. In this way you can isolate the error that caused the abnormal termination.

The backtracking technique does not always work in cases where the error causes infinite looping or in cases where a pointer error leads the program to an incorrect location. In these cases, output found when the program halts is meaningless. Forward tracing is the best method here. Using the debugger, you can advance the program in predetermined steps, examining key locations as you go. When you encounter unexpected output, you can isolate the error that caused it.

1.2 DEBUGGER FEATURES

The debugger is a powerful and flexible tool to help you find errors in your source program.

- It is interactive. You execute debugger commands from your terminal and see their effects immediately.
- It is symbolic. You can refer to program locations by the symbols you used for them in your program.
- It supports many languages. You use the debugger in the language of your source program. If your program is written in more than one language, you can change from one language to another in the course of a debugging session.
- It permits a variety of data forms and types for entry and display. By default, the source language of the program determines the format used for the entry and display of data. You can also impose other formats on the entry and display of data. For example, the contents of a program location may be entered or displayed in hexadecimal, octal, or decimal notation.
- It gives online help. When you type HELP, the debugger responds with a list of commands and related features about which you can receive information during the course of a debugging session.
- In languages that support the feature, it allows for the display of lines of source code during a debugging session. In other words, it allows you to selectively display the programming language statements that make up your program.

The following sections briefly describe some of the available debugger features.

1.2.1 Control of Program Execution

The GO command starts or continues program execution. The program executes until a breakpoint is reached, a watchpoint is modified, an exception condition occurs, or the program terminates.

The STEP command starts or continues program execution, but only as many instructions, lines, or statements are executed as you specify in the parameter of the STEP command. If you specify no parameter, a default value is used.

1.2.2 Breakpoints

By setting breakpoints, you can suspend program execution at specified locations. You can then examine the status of your program by entering debugger commands.

1.2.3 Tracepoints

By setting tracepoints, you can follow the path of program execution. When a tracepoint in your program is activated, the debugger momentarily suspends execution, reports that the tracepoint was reached, and then continues with program execution.

1.2.4 Watchpoints

By setting a watchpoint, you can cause the program to stop whenever a particular variable or other memory area has been modified. When a watchpoint in your program is activated, the debugger suspends execution, and reports the location of the variable modified, the initial and changed values of the variable, and the instruction that caused the modification.

1.2.5 Manipulation of Program Locations and Variables

Using the EXAMINE command, you can determine the value of a variable or program location. Using the DEPOSIT command, you can change that value. Then you can continue program execution to observe the effect of the changed value on program execution, without having to recompile, relink, and rerun the program.

1.2.6 Evaluation of Expressions

Using the EVALUATE command, you can compute the value of source language expressions (whether they be arithmetic, bit string, character string, or Boolean) and of address expressions.

1.2.7 Log Files

If you request, the debugger can record in a log file both the commands you enter during a debugging session and its responses to those commands. You can use log files to keep track of your debugging efforts, or you can use them as command procedures in subsequent debugging sessions.

1.2.8 Command Procedures

A command procedure is a file of debugger commands that you direct the debugger to execute. You can create a command procedure with a text editor, or you can use a debugger log file. You can use command procedures to recreate a debugging session, to continue a previous session, or to save having to type the same debugger commands many times during a debugging session.

1.3 DEBUGGER COMMAND FORMAT

A keyword is a word, group of characters, or other symbolic notation that is recognized within a particular context as possessing special meaning or specifying particular actions. Keywords that the debugger recognizes are called commands, command qualifiers, or command parameters.

In addition, between certain keywords and in accordance with proper syntax, the debugger recognizes user-specified information (also called parameters) such as numbers and file specifications.

The debugger also recognizes characters used as:

- Syntactical delimiters - for example, the semicolon (;), colon (:), comma (,), space ( ), and slash (/)
- Symbols - for example, the period (.), circumflex (^), and backslash (\)
- Operators - for example, the plus sign (+), minus sign (-), and at sign (@)

A debugger command can contain one or more keywords. A debugger command string contains the debugger command, as well as one or more of the following items: qualifier, parameter, DO sequence, or comment.

The following example shows the order in which you specify these items in a command string (optional entries are enclosed in square brackets):

```
command[/qualifier] [parameter] [DO(command string;...)] [!comment]
```

command

The command is the action component of the command string, indicating the specific function to be performed. For example, EXAMINE, CANCEL TRACE, and SET EXCEPTION BREAK are commands.

/qualifier

The qualifier is a keyword that modifies (or qualifies) the action of the command. For example, in SET MODULE/ALL, the qualifier is /ALL. A qualifier must be preceded by a slash.

parameter

A parameter is that which is affected by a commmand, or a value that further modifies the command. For example, in SET TRACE LOOP1, the parameter LOOP1 specifies where the tracepoint is to be set; in SET STEP INTO, the parameter INTO further modifies the command.

DO(command string;...)

A DO command sequence consists of the keyword DO followed by one or more debugger command strings enclosed in parentheses. The DO command sequence is specified in the SET BREAK command and executed at breakpoint activation.

!comment

A comment is a user input string that is preceded by an exclamation point (!). Comments do not affect the execution of debugger commands.

1.3.1 Syntax Rules

You can enter more than one command string on a terminal line. You can also enter more than one command string within a DO sequence. You must separate all command strings with a semicolon.

You can continue a command string on a new terminal line by ending the line with a hyphen (-) and then pressing RETURN. The debugger then prompts you with an underscore (_) on the next terminal line; you can then type in the remainder of your command string.

Spaces are necessary between the following:

- Command and parameter
- Command/qualifier and parameter
- Parameter and DO sequence
- DO sequence and !comment

1.3.2 Abbreviations

Debugger keywords may be abbreviated for ease of entry. A minimal abbreviation for each keyword includes as many characters as needed to make it unique within the set of all keywords.

The following are exceptions to this uniqueness rule:

- Some frequently used commands such as EXAMINE and DEPOSIT may be abbreviated by single characters.
- In some cases, the debugger interprets nonunique abbreviations correctly on the basis of context.

Table 1-1 lists each debugger command, together with all potential command qualifiers, in boldface and lightface type. Characters in boldface represent acceptable abbreviations for that command.

Table 1-1: Debugger Command Abbreviations

@file-spec

CALL

CANCEL **A**LL

CANCEL **B**REAK
- /**A**LL

CANCEL **EXC**EPTION **B**REAK

CANCEL **M**ODE

CANCEL **MODU**LE
- /**A**LL

CANCEL **SC**OPE

CANCEL **SO**URCE
- /**MODU**LE

CANCEL **T**RACE
- /**A**LL
- /**B**RANCH
- /**C**ALL

CANCEL **TY**PE/**O**VERRIDE

CANCEL **W**ATCH
- /**A**LL

CTRL/C

CTRL/Y

CTRL/Z

DEFINE

DEPOSIT
- /**B**YTE
- /**W**ORD
- /**L**ONG
- /**Q**UADWORD
- /**OCTAW**ORD
- /**F**LOAT
- /**D**_FLOAT
- /**G**_FLOAT
- /**H**_FLOAT
- /**A**SCII:n
- /**I**NSTRUCTION
- /**O**CTAL
- /**D**ECIMAL
- /**H**EXADECIMAL

EVALUATE
- /**O**CTAL
- /**D**ECIMAL
- /**H**EXADECIMAL

EVALUATE/**A**DDRESS
- /**O**CTAL
- /**D**ECIMAL
- /**H**EXADECIMAL

EXAMINE
- /**B**YTE
- /**W**ORD
- /**L**ONG
- /**Q**UADWORD
- /**OCTAW**ORD
- /**F**LOAT
- /**D**_FLOAT
- /**G**_FLOAT
- /**H**_FLOAT
- /**A**SCII:n
- /**I**NSTRUCTION
- /**O**CTAL
- /**D**ECIMAL
- /**H**EXADECIMAL
- /**S**YMBOL
- /**N**OSYMBOL
- /**SO**URCE

EXIT

GO

HELP

SEARCH
- /**A**LL
- /**N**EXT
- /**I**DENTIFIER
- /**S**TRING

SET **B**REAK
- [**DO**
- /**A**FTER

SET **EXC**EPTION **B**REAK

SET **LA**NGUAGE

SET **LOG**

SET **MAR**GIN

SET **MAX**_SOURCE_FILES

SET **M**ODE

SET **MODU**LE
- /**A**LL

SET **OU**TPUT

SET **SC**OPE

SET **SEA**RCH

SET **SO**URCE
- /**MODU**LE

SET **ST**EP

SET **T**RACE
- /**B**RANCH
- /**C**ALL

ZK-017/1-82

Table 1-1 (Cont.): Debugger Command Abbreviations

SET TYPE /**O**VERRIDE	**SH**OW **SO**URCE
SET WATCH	**SH**OW **S**TEP
SHOW **B**REAK	**SH**OW **T**RACE
SHOW **C**ALLS	**SH**OW **TY**PE /**O**VERRIDE
SHOW **LA**NGUAGE	**SH**OW **W**ATCH
SHOW **LO**G	**S**TEP /**INS**TRUCTION /**L**INE /**INT**O /**O**VER /**S**YSTEM /**N**OSYSTEM /**SO**URCE /**NOSO**URCE
SHOW **MAR**GIN	**T**YPE
SHOW **MAX**_SOURCE_FILES	
SHOW **M**ODE	
SHOW **MODU**LE	
SHOW **OU**TPUT	
SHOW **SC**OPE	
SHOW **SEA**RCH	

ZK-017/2-82

1.4 BEGINNING A DEBUGGING SESSION

Before you can run your program, you must first compile (or assemble) it, and then link it.

If your program has not yet run and you want to run it with debugger control, specify the /DEBUG command qualifier either when you link it or when you run it.

If your program is already running without debugger control and you want debugger control, interrupt the program (see Section 1.5) and then issue the DCL command DEBUG.

In general, if you intend to debug your program, you should compile and link your program using the /DEBUG command qualifier. In this way, you will be able to debug your program symbolically, making use of symbols used in the source program.

Sections 1.4.1 and 1.4.2 describe how to use the RUN and DEBUG commands, respectively, to start a debugging session. Section 1.4.3 describes how VAX/VMS activates the debugger. Section 1.4.4 describes the debugging environment at start up.

1.4.1 The RUN Command

To start the debugger with the RUN command, you must specify the /DEBUG command qualifier at link time (LINK/DEBUG) or at run time (RUN/DEBUG).

If you specify LINK/DEBUG, control passes to the debugger at run time, providing you do not specify RUN/NODEBUG. If you had specified /DEBUG at compile time, symbol information generated by the compiler is processed by the linker and will be available at run time. You need not specify RUN/DEBUG if you specify LINK/DEBUG.

If you specify RUN/DEBUG but have not specified LINK/DEBUG, control also passes to the debugger, providing you have not specified LINK/NOTRACEBACK. However, if you had specified /DEBUG at compile time, symbol information passed to the linker by the compiler is lost.

In sum, to make use of all available symbolic information, you must specify /DEBUG at compile (or assembly) time as well as at link time. See Section 2.1 for a detailed explanation of how the use of the /DEBUG command qualifier determines the extent of your ability to debug using symbolic information.

The following example demonstrates how to start a debugging session in which you can use the full range of symbolic information in the program MEANSUB:

```
$ BASIC/DEBUG MEANSUB
$ LINK/DEBUG MEANSUB
$ RUN MEANSUB

              VAX-11 DEBUG Version 3.0-3

%DEBUG-I-INITIAL, language is BASIC, module set to 'MEANSUB$MAIN'
DBG>
```

When the debugger gains control, it issues an informational message followed by the DBG> prompt, indicating that it is ready to accept a debugger command.

If you have specified /DEBUG at link time but want your program to run without debugger control, specify /NODEBUG at run time.

1.4.2 The DEBUG Command

If your program is running without debugger control and you want to invoke the debugger, do the following:

- Interrupt the running program by entering CTRL/Y.
- Issue the DEBUG command.

After you enter the DEBUG command, the debugger displays an informational message and the DBG> prompt, indicating that it is ready to accept a debugger command.

You will find the DEBUG command particularly useful when:

- Your program is in an infinite loop. After interrupting your program and entering the DEBUG command, you can debug your program to find the cause of the infinite loop.
- You have entered the RUN/NODEBUG command but later decide that you want debugger control.
- You have not specified the /DEBUG command qualifier at compile time, link time, or run time but want to debug your running program. Note that in this situation, most symbolic information is unavailable to the debugger; you must use virtual memory addresses and not symbols to refer to program and data locations.

After you interrupt your running program and enter the DEBUG command, you will not know at which instruction your program was interrupted. To find out, issue the SHOW CALLS command.

Note that you may enter certain DCL commands after you interrupt your program and still be able to start the debugger by subsequently entering the DEBUG command. See Section 1.5.2 for more information.

The following example demonstrates how to interrupt your program and start the debugger using the DEBUG command. Note that CTRL/Y is echoed as ^Y.

```
$ RUN PROG

^Y

$ DEBUG

                 VAX-11 DEBUG Version 3.0-3

%DEBUG-I-INITIAL, language is COBOL, module set to 'INVENTORY'
DBG>
```

1.4.3 Activation of the Debugger by VAX/VMS

The debugger is a shareable image that VAX/VMS maps into the address space of a user process when SYS$IMGSTA is the first entry in the transfer address array. SYS$IMGSTA is the first entry when you specify any one of the following combinations of DCL commands in the process of program development:

- LINK/DEBUG and not RUN/NODEBUG
- RUN/DEBUG (unless LINK/NOTRACEBACK)
- DEBUG following program interruption (unless LINK/NOTRACEBACK)

When the /NOTRACEBACK qualifier is specified at link time, the linker does not generate a Debug Symbol Table (DST). Since a DST must be available at run time in order for the debugger to execute, specifying /NOTRACEBACK at link time inhibits the debugging of that image regardless of what other qualifiers are or are not specified at link or run time.

At run time, then, given any of the previously mentioned combinations of DCL commands, SYS$IMGSTA, a component of VAX/VMS, maps the debugger (that is, the shareable image DEBUG.EXE) into the highest-addressed portion of P0 space adjacent to the user image and passes control to the debugger.

When the debugger receives control, it initializes a portion of the address space adjacent to the debugger image with three symbol tables -- the Debug Symbol Table (DST), the Global Symbol Table (GST), and the Run-Time Symbol Table (RST). See Sections 2.1.2.1, 2.1.2.2, and 2.1.2.3 for a description of these symbol tables and how the debugger uses them.

Figure 1-1 depicts the layout of the virtual address space of a process that is running a program under debugger control:

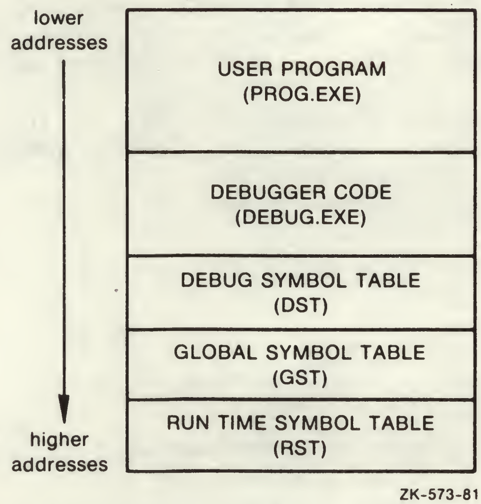

Figure 1-1: Process Address Space Layout

1.4.4 Debugging Environment at Start Up

At start up, the debugger displays an informational message in the following format:

```
                VAX-11 DEBUG      Version Number

    %DEBUG-I-INITIAL, language is xxx, module set to yyy
    DBG>
```

The first line of this message identifies the debugger's software version number. The second line contains initialization information that identifies the current language and program module. The third line contains the debugger prompt DBG>.

1.4.4.1 Language-Dependent Debugging Parameters - Debugging parameters influence how the debugger interprets commands and how it displays the results of command execution.

At start up, the debugger sets a default value for each debugging parameter. These values may be displayed using the appropriate SHOW command, may be altered using the appropriate SET command, and, in some cases, may be canceled using the appropriate CANCEL command.

The default values of some debugging parameters may vary, depending on the current language. That is, a particular parameter may have one default value if the current language is BASIC and another default value if the current language is MACRO. Such parameters are called language-dependent debugging parameters.

The following parameters are language-dependent debugging parameters:

- TYPE
- MODE
- STEP
- OUTPUT

When you change the current language by the SET LANGUAGE command, the default values for language-dependent debugging parameters may change. However, if you have changed the default value of any parameter by means of a SET command, that value remains in effect regardless of a subsequent SET LANGUAGE command.

To find out what the current language is, issue the SHOW LANGUAGE command.

1.4.4.2 Language-Independent Debugging Parameters - At start up, the default values of the following debugging parameters are established independently of the current language setting:

- MODULE
- SCOPE

The module parameter allows for the insertion of a module's symbol information into the Run-Time Symbol Table (RST). At start up, the debugger inserts, into the RST, symbol records for the module containing the transfer address. The name of this module and the name of the language in which the module is written appears in the debugger informational message.

The transfer address is the location within the compiled code specified by the keyword(s) in the source language that signal the beginning of a program. For example, in VAX-11 PASCAL the transfer address is the location within the compiled code specified by the keyword PROGRAM.

The scope parameter influences the debugger's interpretation of symbols.

When you start the debugger with the RUN command, the value of the scope parameter is the module that contains the transfer address. When you start the debugger with the DEBUG command, the value of the scope parameter is the module in which program execution was interrupted.

Chapter 2 contains a full description of how the module and scope parameters affect the debugger's interpretation of symbols.

1.5 INTERRUPTING A DEBUGGING SESSION

Since the debugger is itself an image within the context of the VAX/VMS operating system, it reacts to interruption like other images running in user mode; for instance, the image is interrupted (but unchanged), the terminal type-ahead buffer is purged, and the DCL command interpreter receives control.

To interrupt a debugging session, enter CTRL/Y (echoed as ^Y) as follows:

```
DBG> CTRL/Y

^Y

$
```

The dollar sign prompt ($) indicates that the DCL command interpreter has control.

If you run your program without debugger control, interrupt it, and start the debugger with the DEBUG command, the debugger informational message indicates the name of the module that contains the program location at which program execution was interrupted and the name of the language in which that module was written.

If you run your program with debugger control, interrupt it, and restart the debugger with the DEBUG command, the debugger preserves the module and language information existent at the time of interruption. For example, if before interruption you set language and/or module using the SET LANGUAGE and/or SET MODULE command(s), the parameters specified in these commands remain in effect when the debugger regains control by the DEBUG command.

If the debugger is restarted by the DEBUG command, symbol records from more than one module may be present in the RST.

1.5.1 CTRL/Y and CTRL/C

The effect of issuing CTRL/Y or CTRL/C is the same unless your system or application program contains a routine coded to intercept CTRL/C.

If such a CTRL/C handling routine exists, entering CTRL/C causes control to be passed to the handling routine rather than to the command interpreter.

If a CTRL/C handling routine does not exist, both CTRL/Y and CTRL/C interrupt the debugging session and cause control to be passed to the DCL command interpreter, which signals with the $ prompt.

When the $ prompt is displayed, you can enter any DCL command. Section 1.5.2 discusses the effect of some of these commands in relation to restarting a debugging session.

1.5.2 Options After Interruption

After interruption of a debugging session by CTRL/Y or CTRL/C and the resulting display of the $ prompt, you may want to enter any of the following DCL commands:

- DEBUG - See Section 1.4.2.
- CONTINUE - Passes control back to the debugger or to the program, whichever had control at the time of interruption.
- STOP - Causes abnormal termination of the debugger. The debugger exit handler is not given control.

- EXIT - Causes normal termination of the debugger. The debugger exit handler is given control.
- Commands that are performed within the DCL command interpreter - These commands may be entered and executed without causing termination of the debugger image. For example, after interruption of the debugger by CTRL/Y, if you enter the SHOW DAYTIME command and follow it with the DEBUG command, control will pass back to the debugger. (See Chapter 4 in the VAX/VMS Command Language User's Guide for a list of these commands.)
- Any other DCL command - Causes termination of the debugger. The debugger exit handler is executed.

1.6 ENDING A DEBUGGING SESSION

To end a debugging session, issue the EXIT command or enter CTRL/Z. Both cause normal exit behavior; that is, the debugger exit handler is executed and exit status information is displayed.

The following example demonstrates the ending of a debugging session:

```
DBG>EXIT

$
```

You can also end a debugging session by interrupting with CTRL/Y or CTRL/C, as explained above, and following with certain DCL commands.

CHAPTER 2

SYMBOL REFERENCES AND THEIR INTERPRETATION

In the context of debugging, symbols are strings of characters that represent files, program modules, routines, program locations, variables, arrays, or any other definable units. This chapter explains what you need to know about symbols to debug your program.

2.1 SYMBOLIC DEBUGGING

Symbolic debugging means debugging using symbols, instead of virtual addresses, to refer to memory locations.

For the debugger to interpret symbols, information about these symbols (in the form of symbol records) must be present in the executable image at run time.

You control which symbol records are available at run time by using the /DEBUG command qualifier with the compile and LINK commands.

By default, symbol records for traceback information are available at run time. Traceback information consists of symbol records that describe module names, routine names, and compiler-assigned line numbers. Traceback information is used both by the debugger and by the traceback utility. The traceback utility uses these symbol records to display the call stack when a program terminates abnormally.

Figure 2-1 shows a traceback display of the call stack:

```
%PAS-F-ERRACCFIL, error in accessing file PAS$OUTPUT
%PAS-F-ERROPECRE, error opening/creating file
%RMS-F-FNM, error in file name
%TRACE-F-TRACEBACK, symbolic stack dump follows

module name     routine name     line      rel PC       abs PC

PAS$IO_BASIC    _PAS$CODE                  00000192     00001CED
PAS$IO_BASIC    _PAS$CODE                  0000054D     000020A8
PAS$IO_BASIC    _PAS$CODE                  0000028B     00001DE6
EIGHTQUEENS     EIGHTQUEENS       59       00000020     000005A1
```

Figure 2-1: Traceback Information

To prevent the inclusion of symbol records for traceback information in the executable image, you can specify /NODEBUG with the compile

command or /NOTRACEBACK with the LINK command. Note however that specifying /NOTRACEBACK also disables the debugger. In general, if you intend to debug your program, you want to include traceback information in the executable image. On the other hand, when your program is fully debugged, you might want to prevent the inclusion of traceback information in the executable image in order to reduce the size of the executable image file.

Section 2.1.1 describes how to use the /DEBUG command qualifier to include symbol records in the executable image; Section 2.1.2 describes the symbol tables that contain these records.

However, even if the required symbol records are present in the executable image at run time, the debugger will be able to access them only if they are present in its Run-Time Symbol Table (RST). To put symbol records in the RST, you use the SET MODULE command. Section 2.1.3 describes the SHOW, SET, and CANCEL MODULE commands.

Finally, given that the required symbol records are present in the RST, it might also be necessary to specify the program region (or scope) in which a particular symbol is to be interpreted. For example, symbols with the same name, but with declarations in different routines, must be differentiated from one another. In this case, you must prefix the symbol name with a pathname or use the SET SCOPE command to define the scope. Section 2.3 discusses these topics.

2.1.1 The /DEBUG Command Qualifier

You can specify the /DEBUG command qualifier with one or more of the following commands: a compiler command, the LINK command, the RUN command.

The /DEBUG qualifier allows the debugger to use the following types of symbolic information: local symbols, global symbols, and traceback information (routine names and line numbers).

In this discussion, the term "local symbols" refers to source program symbols as they are declared by the programmer. Some of these symbols may, in fact, have been declared as "global". However, since similar information is stored for these symbols, whether or not they are declared as "global," this distinction is unimportant for the purpose of this discussion. Further, the term "global symbols" is needed to refer to symbols about which the linker stores information in the Global Symbol Table (GST).

If you specify the /DEBUG qualifier at compile time, the compiler includes local symbol information and traceback information in the object module to be used as input to the linker. If you do not specify the /DEBUG qualifier at compile time, traceback information is included in the object module, but not local symbol information.

In some languages, you can control the presence or absence of local symbol information and traceback information in the object module by specifying /DEBUG with or without options at compile time. For those languages that support the options qualifier, the following options are possible:

- To make both traceback and local symbol information available to the debugger, specify /DEBUG or /DEBUG=ALL.
- To deny both traceback and local symbol information to the debugger, specify /NODEBUG or /DEBUG=NONE.

- To make traceback information available, but not local symbol information, specify /DEBUG=TRACEBACK or /DEBUG=NOSYMBOLS.

If you specified the /DEBUG qualifier at compile time, specifying /DEBUG with the LINK command has the following effects:

- Local symbol information is included in the executable image.
- Global symbol information is included in the executable image.
- Traceback information is included in the executable image.
- The image activator is directed to start the debugger at run time.

If you did not specify the /DEBUG qualifier at compile time, specifying /DEBUG with the LINK command has the following effects:

- Traceback information is included in the executable image (unless /NODEBUG was specified with the compile command).
- Global symbol information is included in the executable image.
- The image activator is directed to start the debugger at run time.

If you specified the /DEBUG qualifier at link time, the debugger receives control when you issue a RUN command, whether or not you specify the /DEBUG qualifier at run time.

If you did not specify the /DEBUG qualifier at link time, then you must specify RUN/DEBUG if you want the debugger to receive control. Note that in this case you cannot debug using local symbols.

If you specified LINK/DEBUG but do not want the debugger, specify RUN/NODEBUG.

Table 2-1 shows how specifying the /DEBUG command qualifier with the relevant compile command, LINK command, and RUN command affects the availability of symbolic information in the executable image.

Column 4 in Table 2-1 shows which symbolic information is available in the executable image as a result of specifying the command qualifiers shown in Columns 1 through 3. The following types of symbolic information can be available in the executable image:

- Local symbol information, which consists of symbol records for source program symbols, both "local" and "global." In the executable image, local symbol information is found in the Debug Symbol Table (DST).
- Traceback information, which consists of symbol records for routine names and compiler-assigned line numbers. In the executable image, traceback information is found in the Debug Symbol Table (DST).
- Global symbol information, which consists of symbol records for routine names and global data names. In the executable image, global symbol information is found in the Global Symbol Table (GST).

Table 2-1: The /DEBUG Qualifier and Symbolic Debugging

Compile (or Assembly) Command Qualifier	LINK Command Qualifier	RUN Command Qualifier	Available Symbolic Information
none	none	/DEBUG	traceback
/DEBUG or /DEBUG=ALL	none	/DEBUG	traceback
/NODEBUG or /DEBUG=NONE	none	/DEBUG	none
/DEBUG=TRACEBACK or =(TRACEBACK,NOSYMBOLS)	none	/DEBUG	traceback
none	/DEBUG	none	traceback global symbols
/DEBUG or /DEBUG=ALL	/DEBUG	none	local symbols global symbols traceback
/NODEBUG or /DEBUG=NONE	/DEBUG	none	global symbols
/DEBUG=TRACEBACK or =(TRACEBACK,NOSYMBOLS)	/DEBUG	none	traceback global symbols

2.1.2 Symbol Tables Used by the Debugger

Symbol tables contain symbol records that the debugger uses to associate a symbol with a program location (that is, a virtual address). In addition, by means of some symbol records, the debugger can associate attributes with a program location, such as the length of that location and its data type.

The debugger uses the following three symbol tables:

- Debug Symbol Table (DST)
- Global Symbol Table (GST)
- Run-Time Symbol Table (RST)

2.1.2.1 Debug Symbol Table (DST) - When the linker creates an executable image, it creates a Debug Symbol Table (DST) unless you specify the /NOTRACEBACK qualifier with the LINK command. The linker includes the DST in the executable image.

If you specify the /DEBUG qualifier with the LINK command, the linker includes in the DST all symbol records that are present in the object module(s) from which it creates the executable image. Thus, if symbol records for local symbols are present in the object module(s) (because you specified /DEBUG at compile time), the DST will contain symbol records for local symbols and for traceback information.

If you do not specify the /DEBUG qualifier with the LINK command, the linker includes only symbol records for traceback information. Thus, even if symbol records for local symbols are present in the object module(s), they are not included in the executable image.

A DST symbol record typically contains the name of a symbol, its type, its length, and its value or address. The type information may simply mark the symbol as a routine or a label, or it may define a more or less complex data type. Similarly, the address information may be a simple virtual address or value, or it may specify a more or less complex way of computing the symbol address.

At run time, the debugger uses the symbol records in the DST to build the Run-Time Symbol Table (RST). Section 2.1.2.3 describes the RST.

2.1.2.2 Global Symbol Table (GST) - When the linker creates an executable image, it creates a Global Symbol Table (GST). The linker includes the GST in the executable image only if the /DEBUG qualifier is specified with the LINK command.

The linker creates the GST from its own internal symbol table. As a result, symbol records in the GST contain no more information than the linker needs to do its job. Symbol records in the GST describe all global symbols in the image, such as routine names, procedure entry points, and global data names. Symbol records for these global symbols associate symbol names with virtual addresses or values, but they do not contain information about data types. See the appendix on the VAX-11 object language in the <u>VAX-11 Linker Reference Manual</u> for information on the content and format of these symbol records.

The debugger uses the GST only if it cannot locate needed information in the DST. If a symbol is represented in both the DST and the GST, the DST symbol record usually contains more information about the symbol than the GST symbol record.

2.1.2.3 Run-Time Symbol Table (RST) - When the debugger is activated at run time, it uses available symbol records in the DST and GST to generate symbol entries for the Run-Time Symbol Table (RST).

The RST allows the debugger random access to symbols during a debugging session. Its purpose is to allow efficient access to symbol records contained in the DST and GST. Whenever you mention a symbol in a debugger command, the debugger checks the RST for the information it needs to interpret that symbol. If there is no entry for a symbol in the RST, you cannot access that symbol.

Issue the SET MODULE command in the following format to insert symbol records for one, several, or all modules in the RST:

```
SET MODULE [/qualifier] [module-name [,module-name...]]
```

To include symbol records for one or several modules, specify the module name(s) in the SET MODULE command. In this case, do not specify a command qualifier.

To include symbol records for all modules, specify the /ALL command qualifier in the SET MODULE command. In this case, do not specify a module name or names.

Note that if a parameter in the SET SCOPE command designates a program location in a module whose symbol records are not already in the RST, the debugger copies symbol records of that module into the RST when the SET SCOPE command is executed.

When all the memory space allocated for the RST is occupied by symbol records and you want to include additional symbol records, you must issue the CANCEL MODULE command in the following format:

```
CANCEL MODULE [/qualifier] [module-name [,module-name...]]
```

To delete symbol records of one or several modules, specify the module name or names in the CANCEL MODULE command. In this case, do not specify a command qualifier.

To delete symbol records of all modules, specify the /ALL command qualifier in the CANCEL MODULE command. In this case, do not specify a module name or names.

The following example demonstrates the SHOW, SET, and CANCEL MODULE commands:

```
                    VAX-11 DEBUG Version 3.0-3

%DEBUG-I-INITIAL, language is BASIC, module set to 'MEANSUB$MAIN'

DBG> SHOW MODULE
module name       symbols    language    size
MEANSUB$MAIN      yes        BASIC        172
OTS$LINKAGE       no         MACRO        176
BAS$STOP          no         BLISS        284
BAS$MSGDEF        no         BLISS         68
total modules: 4.                  remaining size: 60784.

DBG> SET MODULE OTS$LINKAGE

DBG> SHOW MODULE
module name       symbols    language    size
MEANSUB$MAIN      yes        BASIC        172
OTS$LINKAGE       yes        MACRO        100
BAS$STOP          no         BLISS        284
BAS$MSGDEF        no         BLISS         68
total modules: 4.                  remaining size: 60676.

DBG> CANCEL MODULE MEANSUB$MAIN

DBG> SHOW MODULE
module name       symbols    language    size
MEANSUB$MAIN      no         BASIC        172
OTS$LINKAGE       yes        MACRO        100
BAS$STOP          no         BLISS        284
BAS$MSGDEF        no         BLISS         68
total modules: 4.                  remaining size: 60856.
```

```
DBG>    SET MODULE/ALL

DBG>    SHOW MODULE
module name        symbols    language    size
MEANSUB$MAIN       yes        BASIC        172
OTS$LINKAGE        yes        MACRO        100
BAS$STOP           yes        BLISS        172
BAS$MGSDEF         yes        BLISS         36
total modules: 4.                  remaining size: 60452.
```

2.2 KINDS OF SYMBOLS

Symbols differ from one another not only in what they represent but also in the contexts in which they can be interpreted.

This section describes:

- Symbols that can always be interpreted in a debugging session
- Symbols that can be interpreted in a debugging session if you define them during the session
- Symbols whose interpretation depends on many factors, such as whether the /DEBUG qualifier was specified at compile time and which modules' symbols are in the run-time symbol table

2.2.1 Debugger Permanent Symbols

Debugger permanent symbols are symbols that are known to the debugger in any debugging context. Thus, you can use them at any time in a debugging session. The following are the debugger permanent symbols and their referents:

- The letter R followed by a number from 0 to 11 represents the corresponding general purpose register. Examples: R0, R1, R2, R3, R4, R5,...R11.
- PC represents the program counter.
- SP represents the stack pointer.
- AP represents the argument pointer.
- FP represents the frame pointer.
- PSL represents the processor status longword.
- The period (.) represents the current entity. You may use it to refer to the program location last referenced by an EXAMINE or DEPOSIT command. See Section 3.2.6 for more information.
- The circumflex (^) represents the logical predecessor of the current entity.
- The backslash (\) represents the value last displayed by an EVALUATE command.

  The backslash is also used in debugger syntax to separate the components of a pathname and the symbol modified by that pathname. Thus, the debugger interprets the backslash according to context.

NOTE

In some languages, the percent character (%) must precede the following debugger permanent symbols; R0 through R11, PC, FP, AP, PSL. Examples: %R0, %R1,...%R11, %PC, %AP, %FP, %PC, %PSL.

In these languages, if the percent character (%) is not present, the debugger interprets these symbols as user-defined program symbols, not as debugger permanent symbols.

2.2.2 Symbols Created by the DEFINE Command

During a debugging session, you can use the DEFINE command to create a new symbol or to change the value of a currently existing symbol.

For example, if you find yourself frequently referencing a difficult-to-remember, nonsymbolic program location, you can define a symbol to represent that program location. Then, for the remainder of the debugging session, you can refer to that program location by the symbol that represents it.

Symbols created with the DEFINE command are destroyed when you terminate the debugging session.

As a result of the following command, MIN1 may be specified during a debugging session instead of PROGRAM\ROUTINEA\BLOCKB\MINIM+20:

```
DBG>DEFINE MIN1 = PROGRAM\ROUTINEA\BLOCKB\MINIM+20
```

See Chapter 9 for a full description of the DEFINE command.

2.2.3 Program Symbols

Program symbols (also called identifiers) are the symbols you use when you write your program.

A program symbol is interpreted according to how and where it is declared and where it is used.

A declaration of a symbol specifies attributes permanently associated with the symbol, such as the data type, how much storage is allocated, and so on. The symbol is interpreted by means of these attributes.

Program symbols fall into one of two categories: (1) data names, which identify data operated on by the program's executable statements and (2) program unit labels and program location labels, which identify or name programs, procedures, lines, or statements.

The following sections discuss the syntactic and semantic differences among program symbols. Syntactically speaking, symbols range from simple characters or names to complex entities composed of several names separated by various delimiting characters.

The following subsections present the basic types of program symbols in order to demonstrate the debugger's capacity to interpret the full range of language-specific symbol syntax.

For those readers who are already familiar with the varieties of program symbols, it may be enough to know that the debugger interprets program symbols in the source-language syntax. Thus, in general, if a symbol is legal in the source language, the debugger is capable of interpreting it.

2.2.3.1 Simple Symbols - The debugger interprets simple symbols according to the syntactical rules of the source language.

A simple symbol is a program symbol that does not contain symbolic prefixes or suffixes that further qualify the data item represented by the symbol.

A simple symbol may refer to an individual data item or to an entire range of data items.

A simple symbol may be a data name that represents, for example, a numeric constant (X), an array (ARR), or a character string (A5$).

If a symbol is legal in the source language, the debugger is generally able to interpret it. For example, if language is set to VAX-11 BASIC, the debugger interprets the symbol T$ as a legal symbol representing a character string.

2.2.3.2 Subscript-Qualified Symbols - The debugger interprets subscript-qualified (or subscripted) symbols in the source language syntax.

A subscripted symbol is a symbol used to refer to a data item in an array.

Because an array consists of an ordered set of data items, you can identify an array item by specifying the array name and the item's position in the array. The position of an array item is indicated by one or more subscripts.

For example, (I+2) is a subscript in the following subscripted symbol:

```
ARR(I+2)
```

The number of subscripts used to identify a data item in an array indicates the dimensionality of the array. Thus, the third member of a one dimensional array named ARR is identified by the subscripted symbol ARR(3), and the array member located in the third row, second column of a two dimensional array named TAB is identified by the subscripted symbol TAB(3,2).

2.2.3.3 Structure-Qualified Symbols - The debugger interprets symbols that identify members of data aggregates according to the syntactical rules of the source language.

A data aggregate that consists of data items of different types is called a data structure or a record. Records contain data items that may contain subitems that may contain sub-subitems, and so on.

For example, the following is a declaration, in VAX-11 PASCAL, of a record Person:

```
TYPE PERSON = RECORD
               NAME : PACKED ARRAY[1..20] OF CHAR;
               AGE  : INTEGER;
               SEX  : (MALE,FEMALE);
               SALARY : INTEGER
               END;

VAR COURTNEY : PERSON;
```

In most languages, to reference a field in a record, you must use a structure-qualified symbol containing the name of the desired field and the names of all other fields that contain that field.

For example, to reference the field Age in the record above, the following structure-qualified symbol is used:

```
COURTNEY.AGE
```

The period (.) is a delimiter used to separate field and record names.

Note that in those languages that permit arrays of records, symbolic references to fields within those records are a combination of subscripted symbol and structure-qualified symbol.

Generally, a legal symbol in the source language is capable of interpretation by the debugger.

2.2.3.4 Pointer-Qualified Symbols

- Some languages use pointer-qualified symbols to reference data items that are not bound by the compiler to specific memory addresses. As with other source-language symbols, the debugger interprets pointer-qualified symbols in the syntax of the source language.

For example, to indicate a pointer type in VAX-11 PASCAL, you specify the name of the base type preceded by a circumflex (^). Thus, if the Record Person in Section 2.2.3.3 is the base type, the pointer variable P is declared as follows:

```
VAR P :  ^PERSON;
```

To access the entire record, the following pointer-qualified symbol is used:

```
P^
```

To access a field within the record, include the name of the field:

```
P^ .SALARY
```

2.3 SYMBOL RESOLUTION IN THE SOURCE LANGUAGE

The recognition or resolution of a symbol during a debugging session depends on the resolution of the symbol in the source language and on the debugger context in which the symbol is mentioned.

This section discusses the factors that influence the resolution of a symbol within the source language. These factors are the context in which the symbol declaration occurs and the presence or absence of a global attribute in the symbol declaration.

2.3.1 Program Context of Symbol Declarations

Typically, programs contain symbol declarations. A symbol declaration associates (or binds) a symbol to an entity and associates with that entity certain properties (or attributes) such as type, storage allocation, and so on. Declarations may be implicit; however, this discussion does not distinguish between implicit and explicit declarations.

The scope of a declaration is the set of program locations wherein the symbol is interpreted as representing the entity bound to it in that declaration.

The same symbol may be declared more than once. Declarations of the same symbol have disjoint or nonoverlapping scopes, and each declaration binds the symbol to a different entity (unless the symbol is global).

If a declaration occurs in a program unit that does not contain other program units, the scope of the declaration is the program unit in which it occurs. If a declaration occurs in a program unit that contains other (nested) program units, the scope of the declaration is the program unit in which the declaration occurs and all other nested program units that do not themselves contain declarations of the same symbol.

Figure 2-2 shows nested program units containing several symbol declarations. The symbol X is declared in two program units. The scope of X declared in A is A, B, and D, but not C, because X is redeclared in C. The scope of X declared in C is C; the scope of Y is B and C; and the scope of Z is D.

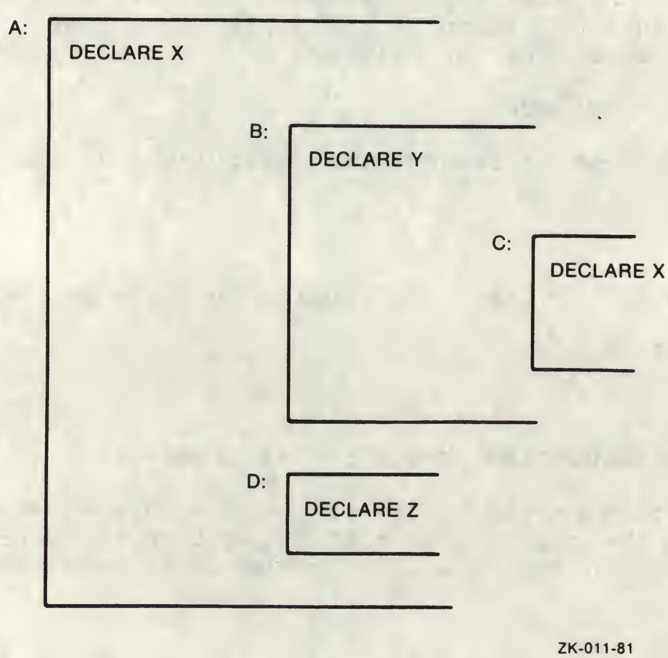

Figure 2-2: Scope of Symbol Declarations

2.3.2 Global Symbols

A global symbol is a symbol whose declaration contains a global attribute.

The presence or absence of a global attribute in a declaration influences the resolution of that symbol.

Global symbols make possible references to the same entity by program units that do not contain one another (nonnested program units). To accomplish this, each program unit declares the symbol with the global attribute, though not necessarily with all other attributes identical.

Thus, a global symbol is bound by several declarations to the same entity. This contrasts with multiple declarations of the same local (or nonglobal) symbol where each declaration associates the symbol with a different entity.

Strictly speaking, the scope of the declaration of a global symbol is not different from the scope of the declaration of a nonglobal symbol; however, the entity referred to by that symbol may be capable of interpretation outside of the scope of its declaration in any one program unit, namely, in those other program units that also declare it.

Figure 2-3 depicts declarations of global symbol X in two nonnested program units. Both program units also make references to X. Because X is declared with the global attribute in both program units, references to X in either program unit are references to an identical memory location, the entity represented by X.

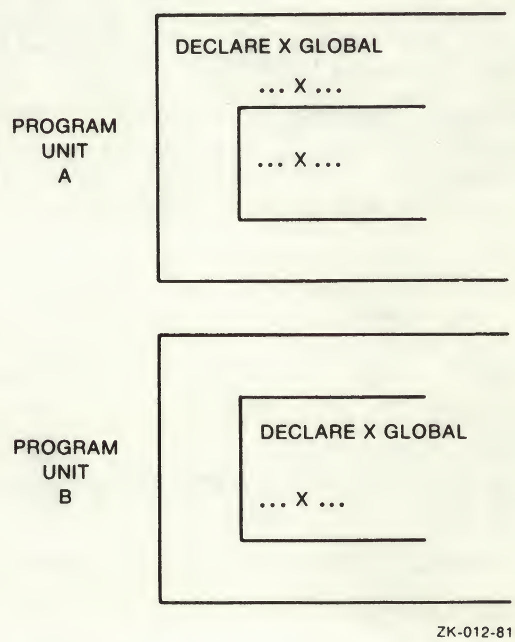

Figure 2-3: Global Symbol X

2.4 SYMBOL RESOLUTION IN THE DEBUGGER

As described in Section 2.3.1, in the source language, a symbol is interpreted as representing the entity associated with it in that declaration within whose scope the symbol occurs.

In debugging, however, wherein random access to symbols anywhere in the program may be necessary, you use pathnames to distinguish among multiple declarations of the same symbol.

A pathname consists of one or more program location labels that serve to specify a program location or range of program locations. Hence, a pathname is used to "locate" a symbol within the scope of the declaration that binds it to the entity you want to reference. Of course, the program location(s) indicated by the pathname must be within the scope of some declaration of the symbol; otherwise the symbol cannot be interpreted at all.

Thus, when you are debugging a program containing multiple declarations of the same symbol and you want to indicate to the debugger that you wish the symbol to be interpreted as representing one of several possible entities, you simply specify a pathname with the symbol. The debugger then interprets the symbol as representing the entity associated with it in the declaration whose scope includes the program location(s) denoted by the pathname.

If you specify a pathname as a parameter in the SET SCOPE command, you are in effect establishing that pathname as a default pathname prefix, to be used in all symbol references that do not already contain a pathname prefix. See Section 2.4.2 for more information on the SET SCOPE command.

You may specify as many pathname parameters as you like in the SET SCOPE command; the debugger attempts to interpret the symbol using the first pathname listed. If that fails, the debugger uses the second pathname and continues in this manner until it finds a pathname that identifies a program location within the scope of a declaration of the symbol. The debugger then interprets the symbol as representing the entity identified in the declaration.

If you do not specify a pathname prefix when you use a symbol in a debugger command and you have not specified default pathnames using the SET SCOPE command, the debugger uses the program location identified by the current PC as a default pathname prefix.

When you use a symbol X in a debugger command, you may or may not use a pathname prefix. If you use a pathname prefix, the debugger interprets X as if it appeared in the program location(s) defined by that pathname. The debugger issues an error message if there is no declaration of X whose scope includes the program location(s) defined by the pathname. If you do not use a pathname prefix, the debugger attempts to interpret X using default pathname(s), either (1) pathname(s) you specified in the SET SCOPE command, or (2) if you have not specified a pathname, the pathname indicated by the program location containing the current PC.

By way of example, assume you enter the following two commands:

```
DBG> SET SCOPE A\B,C
DBG> EXAMINE X
```

You have not specified a pathname prefix in the EXAMINE command. Consequently, the debugger uses A\B\ as a pathname prefix and attempts to locate a declaration of X whose scope includes the program location identified by the pathname A\B\. If the debugger finds such a declaration of X, it uses that declaration to interpret X in the EXAMINE command. If it does not find such a declaration of X, the debugger then repeats the procedure using C\ as a pathname prefix.

If the debugger fails to find a declaration of X whose scope includes the program locations identified by the default pathnames, it searches the entire Run-Time Symbol Table (RST) for a declaration of X. If the debugger locates one, it resolves X using that declaration. If it locates more than one declaration of X, it issues an error message to the effect that an ambiguous reference has been made.

If the debugger does not find a declaration of X in the Run-Time Symbol Table (RST), it searches the Global Symbol Table (GST). If the debugger does not find declaration of X in the RST, it displays a message indicating that the symbol could not be found. See Section 2.1.2.3 for information on what to do.

To utilize fully the debugger's capacity for distinguishing among multiple declarations of the same symbol, you must be thoroughly familiar with pathname specification.

2.4.1 Pathname Specification

A pathname is a string of program location labels that identifies a program location or a range of program locations. A pathname is used in two contexts:

- As a prefix to a symbol in a debugger command
- As a parameter in the SET SCOPE command

The labels in a pathname are strung together so that each label designates a program unit that contains the program unit designated by any label to its right. Thus, if PROG is a label appearing in a pathname, labels to the left designate "containing" program units, while those to the right designate "contained" program units.

The backslash character (\) is used both to separate pathname elements from one another and to separate the entire pathname from the symbol to which it is prefixed. Note that if the pathname is specified as a parameter in the SET SCOPE command, it is not followed by a symbol; therefore, the rightmost pathname element in the pathname is not followed by a backslash. On the other hand, if the same pathname is used to prefix a particular symbol in a command, the rightmost pathname element is followed by a backslash to separate it from the symbol.

The kinds of labels used in a pathname vary somewhat from language to language; however, in general, a pathname always includes a module name and usually includes one or more of the following elements:

- Routine name(s)
- Block name(s)
- Invocation number
- Line number
- Numeric label

A routine is a separately invocable program unit, that is, a program unit that may be activated by a call. A block is a program unit that is activated in normal execution sequence (inline); it is not separately invocable. A routine may contain one or more blocks and/or routines. A block may contain one or more routines and/or blocks. That is, there may be nesting of blocks and routines.

The following is a typical pathname consisting of module, block, and routine names:

```
INV\PARTS\AUTO\
```

In languages that use line numbers, the %LINE pathname element specifies the source-code line. The following is a pathname that uses the %LINE element:

```
MEDIAN\%LINE 330\
```

In languages that allow recursion, an invocation number is used in a pathname to distinquish among multiple invocations of the same program unit (and therefore among multiple generations of symbols). An invocation number is always associated with a routine name, never a block name; one or more spaces must separate a routine name from an invocation number. (See Section 2.4.1.2 on invocation numbers for more information.) The following is a pathname that specifies an invocation number:

```
MOD\ROUT 1\BLK\
```

Figure 2-4 shows a program containing nested routines and blocks with declarations of the symbol X. Examples 1 through 4 refer to Figure 2-4. In these examples, pathnames are used as prefixes of the symbol X in the EXAMINE command and as parameters in the SET SCOPE command.

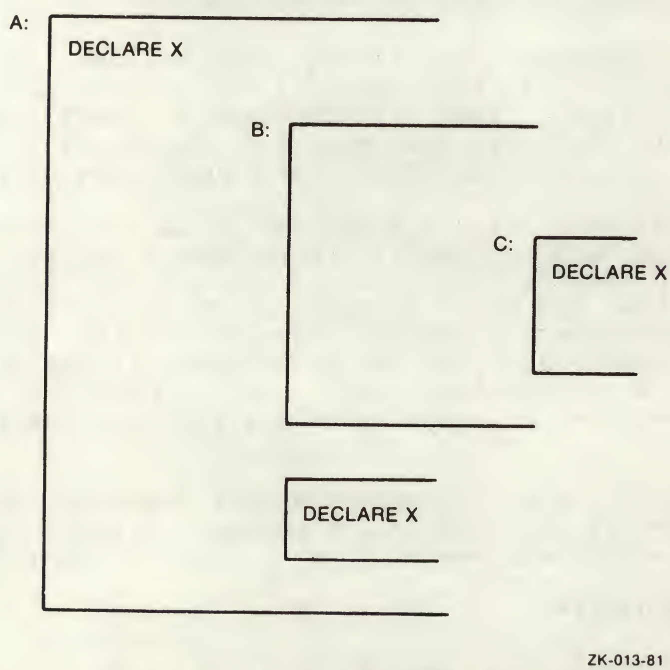

Figure 2-4: Pathnames and Scope

Example 1

```
DBG> EXAMINE A\X
```

The pathname prefix A\ identifies the range of program locations defined by the program unit A. The debugger attempts to locate a declaration of X whose scope includes A. Since such a declaration of X appears in A, the debugger interprets X as representing the entity declared in A. The debugger then examines the value of X.

The scope of the declaration of X in A consists both of those locations in A that are not also in other blocks and of B. C and the unnamed (anonymous) block are not in the scope of the declaration of X in A because they also contain declarations of X.

Example 2

```
DBG> EXAMINE A\B\C\X
```

The pathname prefix A\B\C\ identifies program unit C. The debugger searches for a declaration of X whose scope includes C. It finds such a declaration in C and interprets X as representing the entity declared in C. The debugger then examines the value of X.

Remember that a symbol redeclared in a contained program unit is interpreted according to the declaration in that unit, not the declaration in the outer unit. Thus, the symbol X in C has a different meaning (as declared) than the symbol X that is declared in A.

Example 3

```
DBG> EXAMINE A\%LINE 110\X
```

The pathname A\%LINE 110\ identifies a program location within the anonymous block because line number 110 is contained in the anonymous block. The debugger searches for a declaration of X whose scope includes line number 110. It finds such a declaration in the anonymous block and interprets X as representing the entity declared therein. The debugger then examines the value of X.

Note that only by means of the %LINE pathname element is it possible to identify program locations in an anonymous block. Thus, in this example, without using the %LINE pathname element, it would be impossible to reference the entity represented by X as declared in the anonymous block.

Example 4

```
DBG> SET SCOPE A\B
DBG> EXAMINE X
```

Since no pathname prefix is used in the EXAMINE command, the default pathname prefix A\B\ is attached to X. The debugger searches for a declaration of X whose scope includes B, the range of locations specified by A\B\. It locates such a declaration in A and interprets X as representing the entity declared in A. The debugger then examines the value of X.

2.4.1.1 Pathname Completion - In programs where deep nesting of program units occurs, the naming of every containing program unit in a pathname can be burdensome. To make it easier for you to specify pathnames, the debugger supports pathname completion.

A complete pathname is a pathname that mentions every pathname element: all routines and blocks, as well as label and line numbers, if applicable. An incomplete or abbreviated pathname is a pathname that does not mention every pathname element in the specification.

When the debugger encounters a pathname, it first determines whether the pathname is complete or incomplete. If it is complete, the

debugger uses the pathname to resolve the symbol reference. If it is incomplete, the debugger determines whether it is an abbreviation for a complete pathname and, if so, whether it is an unambiguous abbreviation.

An incomplete pathname is an abbreviation for a complete pathname if:

1. The pathname elements in the incomplete pathname appear in the complete pathname in the same order.
2. The rightmost element in the complete pathname is the rightmost element in the incomplete pathname.

For example, the incomplete pathname

    RUG\%LINE 784\

is an abbreviation for the complete pathname

    MAT\CLR1\RUG\%LINE 784\

because (1) RUG\ and %LINE 784\ in the incomplete pathname are in the same order as in the complete pathname and (2) the rightmost element %LINE 784\ in the complete pathname is the rightmost element in the incomplete pathname.

The abbreviated pathname is unambiguous if there is not more than one complete pathname for which it is an abbreviation.

For example, let us assume that two different subroutines, both named SUB, are declared in program units PROG and QUAR, respectively. The incomplete pathname SUB is ambiguous because there are two complete pathnames, PROG\SUB and QUAR\SUB, for which it is an abbreviation.

An incomplete pathname is acceptable to the debugger so long as it is an abbreviation (according to the definition above) for one (and not more than one) complete pathname.

Unique symbols can always be specified without pathname qualification.

Global symbols may be specified by preceding the symbol with a backslash (\). For example, \X indicates that X is a global symbol.

2.4.1.2 Invocation Numbers - Routines are separately invocable program units; that is, they are activated by calls, rather than by inline execution. Thus during the execution of a program, there may be several simultaneous invocations of a routine or none at all.

If a symbol representing a dynamic entity is declared in a routine or in a block within a routine, the entity represented by that symbol is generated anew each time the routine is invoked. For each invocation of the routine, there is a corresponding generation of the entity. Invocation numbers are used in pathnames to denote particular routine invocations (and thus particular generations of an entity).

Invocation numbers are nonnegative decimal integers inserted in the pathname following the name of the rightmost routine in the complete pathname. The number zero (0) denotes the most recent invocation of the innermost (most deeply nested) routine; the number one (1) denotes the invocation before that; and so on.

For example, if a module MOD contains a routine ROUT which contains a block BLK, then the pathname that identifies the generation of BLK that resulted from the most recent invocation of ROUT is:

```
MOD\ROUT 0\BLK
```

The pathname that identifies the generation of BLK that resulted from the previous invocation of ROUT is:

```
MOD\ROUT 1\BLK
```

Every complete pathname that contains the name of a separately invocable entity has an invocation number. When an invocation number is not present in a pathname that contains the name of a routine, the debugger implicitly assumes the most recent invocation of the routine and supplies the default invocation value zero (0).

Every pathname containing the name of more than one routine specifies an invocation of the innermost routine. In a pathname, invocation numbers cannot be associated with a routine name that appears to the left of another routine name.

How you use a pathname with an invocation number to specify a particular generation of a dynamic entity depends on the relative positions of the innermost routine and the program unit containing the declaration of the entity. The symbol bound to the dynamic entity you want to reference may be declared in any of the following:

1. The innermost routine

2. A block contained in the innermost routine

3. A program unit that contains the innermost routine

These three possibilities give rise to the three program situations illustrated in Figures 2-5, 2-6, and 2-7.

1. **Symbol Declared in the Innermost Routine**

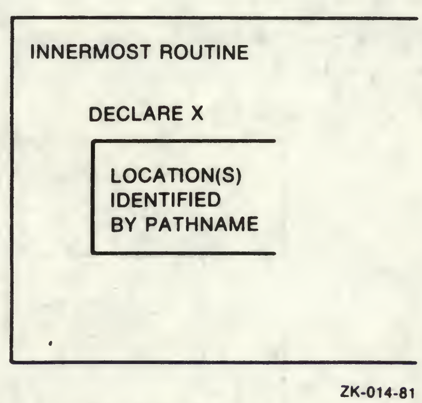

Figure 2-5: Declaration in the Innermost Routine

If the symbol is declared in the innermost routine, the debugger references the generation of the entity corresponding to the invocation of the routine (if there is one) denoted by the pathname. Figure 2-5 illustrates this program situation.

2. **Symbol Declared in a Block Contained in the Innermost Routine**

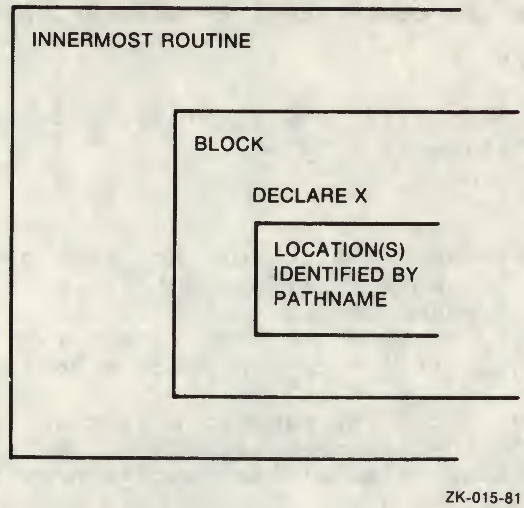

Figure 2-6: Declaration in a Contained Block

If the symbol is declared in a block contained in the innermost routine and if program execution has reached that block, then the debugger references the generation of the entity corresponding to that block activation.

However, if program execution has not reached the block containing the declaration (as would be the case, for example, if you set a breakpoint at a location in the routine prior to the location that marks the beginning of the block), then the generation of the symbol specified by the pathname does not exist, and the debugger issues an error message to that effect. Figure 2-6 illustrates this program situation.

3. **Symbol Declared in a Program Unit That Contains the Innermost Routine**

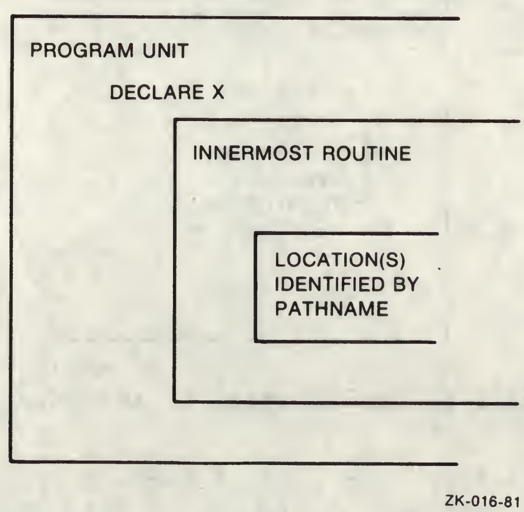

Figure 2-7: Declaration in a Containing Program Unit

Some invocation of a program unit provides the context for the invocation of the innermost routine denoted by the pathname. The

debugger references the generation of the entity corresponding to that invocation of the program unit. Figure 2-7 illustrates this program situation.

Note that when you specify an invocation number, you must leave one or more spaces or tabs between it and the separately invocable routine.

2.4.2 The **SET**, **SHOW**, and **CANCEL** SCOPE Commands

The purpose of the SET SCOPE command is to indicate one or more pathname prefixes to be used in the interpretation of symbols without pathname prefixes.

The numbers 0, 1, 2, and so on, which appear in pathnames as invocation numbers, may also be used as parameters in the SET SCOPE command. Used in this way, these numbers are numeric pathnames.

The numeric pathname 0 specifies that symbols without pathname prefixes are to be interpreted as if they appeared in the currently active routine; numeric pathname 1, in the program unit that contains the call to the currently active routine; numeric pathname 2, in the program unit that contains the call to the program unit that contains the call to the currently active routine; and so on.

If you do not issue a SET SCOPE command, the debugger exhibits a default behavior equivalent to the SET SCOPE 0 command. Thus, by default, the debugger interprets all symbols that do not have pathname prefixes as if they appeared in the currently active routine (specified by the numeric pathname 0).

You can set up a symbol scope search list by specifying more than one parameter in the SET SCOPE command. The debugger uses the first parameter specified as a pathname prefix to interpret a symbol without a pathname prefix. If this interpretation fails, the debugger uses the next parameter listed in the SET SCOPE command in a similar fashion and continues until it successfully interprets the symbol or until it exhausts the parameters specified in the SET SCOPE command.

The following is a list of acceptable parameters to the SET SCOPE command:

- MODULE\ROUTINE\BLOCK

  This is a legal pathname. Pathnames may be complete or incomplete. Note that there may be one or more routines and/or blocks.

- 0, 1, 2, 3, and so on

  These numbers are numeric pathnames. The numeric pathname 0 specifies that symbols without pathname prefixes are to be interpreted as if they appeared in the currently active routine; numeric pathname 1, in the program unit that contains the call to the currently active routine; numeric pathname 2, in the program unit that contains the call to the program unit that contains the call to the currently active routine; and so on.

- \

  The backslash (\) specifies that a symbol without a pathname prefix is to be interpreted as a global symbol.

Note that the debugger always distinguishes between the module-level scope within a module and the routine-level scope within a module. In those languages (such as BLISS) where data can be declared at both the module level and at the routine level, the distinction between the module-level scope and the routine-level scope is quite clear to the programmer. That is, the programmer is aware of which data names are declared at the module level and which ones are declared at the routine level, and can therefore specify the correct scope for the data names used.

On the other hand, in some languages (such as FORTRAN and COBOL), data can be declared only at the routine level. At the module level, only one data name is declared, the routine name. For example, a program PROG in such a language consists of a module PROG within which there is a declaration of a routine PROG within which there are data declarations.

From the debugger's point of view, the command SET SCOPE PROG in a language such as FORTRAN or COBOL sets the scope to the module level. Since the programmer is typically interested in data declarations at the routine level, this command does not produce the desired result. In this case, the programmer must issue the command SET SCOPE PROG\PROG to set scope to the routine level where the data names of interest are declared.

The following is an example of the SET SCOPE command:

```
DBG>SET SCOPE 1, MODB, \
```

This command establishes a symbol scope search list. As a result of this command, the debugger attempts to interpret a symbol without a pathname prefix as if (1) it appeared in the program unit that contains the call to the currently active routine, (2) it appeared in MODB, (3) it were a global symbol.

If you want to examine the current parameters in effect, issue the command:

```
DBG>SHOW SCOPE
```

If you want to change the existing scope search list, issue a SET SCOPE command specifying the desired parameters.

If you want to cancel the existing scope search list, issue the command:

```
DBG>CANCEL SCOPE
```

The CANCEL SCOPE command has the same effect as the SET SCOPE 0 command.

CHAPTER 3

REFERENCING PROGRAM LOCATIONS

In debugging, you make references to program locations in order to stop program execution, to examine and deposit values, and to set breakpoints, watchpoints, and tracepoints.

All references to program locations are called address expressions. An address expression may be a simple address or an expression consisting of one or more simple addresses, operators, and delimiters.

Generally, when the debugger interprets an address expression, the results are a program location and a type that is associated with the contents of that location.

The first section of this chapter contains a discussion of type; the second section, a description of various simple addresses; and the third section, a discussion of the use of operators and delimiters to form expressions.

3.1 TYPE

When a symbol is declared in the source language, the language compiler associates a type with the entity that the symbol represents. Thereafter, the entity is interpreted in that language-dependent (or compiler-generated) type. In general, the debugger understands language-dependent types that are associated with entities by declaration in the source language.

However, in debugging, because all references are not symbolic, an entity may not have a compiler-generated type. In this case, the entity is interpreted using a debugger default type. The default type is used only when the entity has no type associated with it already.

The following are the debugger types:

- BYTE, which designates the byte integer type (length 1 byte)
- WORD, which designates the word integer type (length 2 bytes)
- LONGWORD, which designates the longword integer type (length 4 bytes)
- QUADWORD, which designates the quadword integer type (length 8 bytes), whose values are signed integers in the range -2**63 to 2**63 -1
- OCTAWORD, which designates the octaword integer type (length 16 bytes), whose values are signed integers in the range -2**127 to 2**127 -1

- FLOAT, which designates the F_floating type (length 4 bytes), whose values may range from .29*10**-38 to 1.7*10**38 with approximately 7 decimal digits precision
- D_FLOAT, which designates the D_floating type (length 8 bytes), whose value range is the same as the F_floating type but with approximately 16 decimal digits precision
- G_FLOAT, which designates the G_floating type (length 8 bytes), whose values may range from .56*10**-308 to .9*10**308 with approximately 15 decimal digits precision
- H_FLOAT, which designates the H_floating type (length 16 bytes), whose values may range from .84*10**-4932 to .59*10**4932 with approximately 33 decimal digits precision
- ASCII:n, which designates the ASCII character string type (length n bytes)
- INSTRUCTION, which designates the VAX-11 instruction type (variable length)

To change the default type, specify the desired type as a parameter in the SET TYPE command.

For example, the following command changes the default type to 6-byte ASCII string:

```
DBG> SET TYPE ASCII:6
```

As a result of this command, the debugger interprets any entity without a compiler-generated type as a 6-byte ASCII string.

To see what the current default type is, issue the command:

```
DBG> SHOW TYPE
```

If you want all entities, even those with compiler-generated types, to be interpreted in one of the above types, you establish an override type by specifying the desired type as a parameter in the SET TYPE/OVERRIDE command.

For example, the following command directs the debugger to interpret all entities, even those with compiler-generated types, as VAX-11 instructions:

```
DBG> SET TYPE/OVERRIDE INSTRUCTION
```

To cancel an override type, issue the command:

```
DBG> CANCEL TYPE/OVERRIDE
```

To see what override type is in effect, issue the command:

```
DBG> SHOW TYPE/OVERRIDE
```

If you want the debugger to interpret an entity in one of the above types, but only for the duration of the command that mentions the entity, specify the desired type as a command qualifier. A type specified as a command qualifier is a command override type. For example, the following command directs the debugger to interpret the entity RADIUS as a byte integer:

```
DBG> EXAMINE/BYTE RADIUS
```

As a result of this command, any associated compiler-generated type or current override type established by the SET TYPE/OVERRIDE command is overriden for the duration of the command.

To sum up, you can control the type used by the debugger to interpret an entity in three ways. In increasing order of power, these are:

- Set the default type by the SET TYPE command
- Set an override type by the SET TYPE/OVERRIDE command
- Specify a command override type by using a type command qualifier

Thus, a command override type overrides any compiler-generated type, default type, or override type; an override type overrides any compiler-generated type or default type; a compiler-generated type overrides the default type; and the default type, being the weakest type, overrides no other type.

See Chapter 4 and Chapter 7 for examples of how these types are used in debugging.

3.1.1 The Type Associated With Address Expressions

The type associated with an address expression depends on the current default type, any override types currently in effect, and on the address expression itself.

The debugger associates a type with an address expression in the following way:

1. If an address expression is found in a command that contains a type command qualifier, then the type specified by the qualifier is associated with the address expression.

2. If an address expression is found in a command that does not contain a type command qualifier, then any override type established by the SET TYPE/OVERRIDE command is associated with the address expression.

3. If a type is not associated with the address expression in one of the above ways, then the type associated with the address expression depends on the address expression itself.

   - An address expression that consists of a single, symbolic reference has the type associated with that reference by the source language (the compiler-generated types).

   - The debugger symbols for current entity, logical predecessor, and logical successor have the type associated with the address expression for which they are an abbreviation.

4. If a type is not associated with the address expression in one of the above ways, then the address expression is given the default type.

3.2 SIMPLE ADDRESSES

In debugging, you can make references to program locations using symbolic notation present in your program, as well as other notation generated by the language compiler and notation peculiar to the debugger itself. Taken as a whole, these forms of notation are called simple addresses.

This section presents each form of simple address and explains how the debugger interprets it.

3.2.1 Symbolic References

Symbolic references are program location references that use program symbols, with or without pathname prefixes.

A symbolic reference may be a source-language symbol or a symbol created with the debugger DEFINE command.

In the following examples, the symbolic references DAT1 and ROUTINE\BLOCK\DAT1 are simple addresses in the debugger EXAMINE command:

```
DBG> EXAMINE DAT1

DBG> EXAMINE ROUTINE\BLOCK\DAT1
```

Both commands direct the debugger to display the value of DAT1 in its associated type.

3.2.2 Line Numbers

Some language compilers generate line numbers during the compilation process to facilitate references by the compiler to program locations.

In a debugging session, you can use these line numbers to refer to lines of code in your program. When the debugger encounters the %LINE symbol, it interprets the following number as a compiler-generated line number and uses it as a simple address to reference that location.

The following example shows how the %LINE symbol, together with a line number, is used as a simple address:

```
DBG> SET BREAK %LINE 232
```

To use line numbers as simple addresses, it is helpful to have a compiler listing which, among other things, lists each line of code with its corresponding line number. You obtain a listing by specifying the /LIST qualifier at compile time.

You may also use line numbers with pathname qualification as simple addresses. The syntax for specifying a pathname-qualified line number may take one of two forms:

- The pathname precedes the entire line number notation. For example, MODULE\%LINE 30
- The pathname is inserted between %LINE and the line number itself. For example, %LINE MODULE\30

The %LINE symbol can be used to identify anonymous blocks. See Section 2.4.1 for an example of the %LINE symbol used in this way and for more information about the %LINE symbol.

3.2.3 Statement Numbers

In those languages that allow more than one statement on a line, statement numbers are used to differentiate among statements on the same line.

A statement number consists of a line number, followed by a period and a number indicating the statement. The format is as follows:

```
%LINE xxx.yy
```

In this format, xxx is the line number and yy is a number specifying the statement. The number one (1) represents the first statement that begins on the line; the number two (2), the second; and so on.

For example, in VAX-11 BASIC, the second statement on line 500 is expressed as follows:

```
%LINE 500.2
```

You may also use statement numbers with pathname qualification as simple addresses. The syntax for specifying a pathname-qualified statement number may take either of the following two forms:

- The pathname precedes the entire statement number notation. For example, MODULE\%LINE 30.3
- The pathname is inserted between %LINE and the number itself. For example, %LINE MODULE\30.3

3.2.4 Numeric Labels

In some languages, numeric labels are used to label lines of source code in a program. You can use these labels to reference lines of code in a debugging session by prefixing them with the %LABEL symbol.

The format of a reference to a numeric label in the debugger syntax is

```
%LABEL xxx
```

Here, xxx is a user-defined numeric label.

For example, in VAX-11 FORTRAN, a numeric label is a number you place in a designated column of your program at a strategic program location so that you can reference that location. In debugging your program, you must prefix a reference to a numeric label with the %LABEL symbol.

Figure 3-1 shows part of a compiler listing of a VAX-11 FORTRAN program. The numbers in the leftmost column are line numbers; you refer to them in a debugging session using the %LINE prefix. The numbers in the next column to the right are numeric labels; you refer to them in a debugging session using the %LABEL prefix.

```
          C     PROGRAM TO FIND THE AREA
          C     OF A CIRCLE

0001            PROGRAM CIRCLE
0002      1     TYPE 5
0003      5     FORMAT (' ENTER RADIUS VALUE ')
0004            ACCEPT 10,RADIUS
0005      10    FORMAT (F6.2)
0006                 IF (RADIUS .LE.0) GO TO 20
0007            PI = 3.1415927
0008            AREA = PI*RADIUS**2
0009            TYPE 15,AREA
0010      15    FORMAT (' AREA OF CIRCLE EQUALS ',F10.3)
0011                 GO TO 30
0012      20         TYPE 25
0013      25         FORMAT (' PLEASE TYPE A POSITIVE NUMBER')
0014                 GO TO 1
0015      30            STOP
0016            END
```

Figure 3-1: Line Numbers and Numeric Labels

The following debugger command halts program execution at numeric label 20 in the VAX-11 FORTRAN program shown in Figure 3-1:

```
DBG>SET BREAK %LABEL 20
```

Note that program execution is stopped at the same program location if the following command is issued:

```
DBG>SET BREAK %LINE 12
```

You can use a numeric label with pathname qualification as a simple address. The syntax for specifying a pathname-qualified numeric label may take one of two forms:

- The pathname precedes the entire numeric label notation. For example, CIRCLE\%LABEL 10
- The pathname is inserted between %LABEL and the number itself. For example, %LABEL CIRCLE\10

3.2.5 Literals

A literal is any character string in the source language that is a constant but is not a symbol.

Numeric literals used as simple addresses denote virtual memory addresses; they may also be used in address expressions as offsets from program locations, in which case they may be positive or negative numbers to indicate offset direction.

In the following debugger command, the literal 4 denotes a 4-byte offset from the memory address represented by OPT. This debugger command deposits the value of X, which is to be interpreted as a byte integer, in the program location specified by the address expression OPT+4.

```
DBG>DEPOSIT/BYTE  OPT+4 = X
```

Numeric literals are frequently used to denote virtual memory addresses, as in the following example:

```
DBG> EXAMINE 5032
```

In the evaluation of an address expression containing a literal of a type other than integer (such as floating point, bit string, or complex), the debugger attempts to convert that literal into integer form in the semantics of the source language. If the source language supports such conversion, the debugger uses the resulting integer value; if the conversion is not supported, the debugger issues an error message.

Note that in some languages you may use a radix operator with any legal, source-language numeric literal if you want that literal to be interpreted in decimal (D), octal (O), or hexadecimal (X) radix. See Section 7.2.1.1 for more information.

3.2.6 Current Entity Symbol (.)

The period (.) is the current entity symbol. You may use it to make a reference to the program location last referenced by an EXAMINE or DEPOSIT command.

In other words, whenever either of the following two commands is executed, the value of the current entity is set equal to the value of the address-expression in that command:

- EXAMINE address-expression
- DEPOSIT address-expression = value

Further, the debugger interprets the value of the current entity in the type associated with the address-expression in the EXAMINE or DEPOSIT command that set the current entity.

In the example below, EXAMINE RADIUS sets the current entity. The command "EXAMINE ." results in a display of the name of the current entity (CIRCLE\RADIUS) and the value of the current entity in the floating point type.

```
DBG> EXAMINE RADIUS                            !Sets current entity
CIRCLE\RADIUS:      0.0000000E+00              !symbol.

DBG> EXAMINE                                   !Current entity is
CIRCLE\RADIUS:      0.0000000E+00              !RADIUS.  Display type
                                               !used is the type of
                                               !RADIUS.

DBG> DEPOSIT PI = 3.141593                     !Sets current entity
                                               !symbol.

DBG> EXAMINE
CIRCLE\PI:        3.141593                     !Current entity is PI.
```

3.2.7 Logical Predecessor Symbol (^)

The circumflex (^) is the logical predecessor symbol. When the current entity symbol refers to an entity in an aggregate such as an array, you may use the logical predecessor symbol to refer to that entity in the aggregate that is logically prior to the current entity.

The logical predecessor of an entity may not be its physical predecessor. That is, the logical predecessor may not occupy the region of physical storage directly preceding that of the current entity. Such is the case, for example, when the current entity refers to the cell of a disconnected array A(2) whose logical predecessor A(1) is not stored physically adjacent to A(2) in memory. On the other hand, the logical predecessor of an entity may also be its physical predecessor. For instance, in a connected array, A(1) is both the logical and physical predecessor of A(2).

In some cases the current entity may not have a logical predecessor. For instance, the first cell of an array A(1) has no logical predecessor, and hence its physical predecessor is logically unrelated to it.

As with the current entity, the debugger uses type and symbolic information associated with the logical predecessor.

The following example demonstrates how to use the logical predecessor symbol to examine the cells of an array:

```
DBG> DEPOSIT CHAR(1) = '1234567890'  !Sets current entity symbol.

DBG> DEPOSIT CHAR(2) = 'ABCDEFGHIJ'  !Sets current entity symbol.

DBG> DEPOSIT CHAR(3) = 'abcdefghij'  !Sets current entity symbol.

DBG> EXAMINE .                       !Examines current entity
MOD\CHAR(3):   abcdefghij            !which is CHAR(3).

DBG> EXAMINE ^                       !Logical predecessor is
MOD\CHAR(2):   ABCDEFGHIJ            !previous array member
                                     !CHAR(2).

DBG> EXAMINE ^                       !Logical predecessor is
MOD\CHAR(1):   1234567890            !previous array member
                                     !CHAR(1).
```

3.2.8 Logical Successor Symbol (RETURN)

The RETURN key is the logical successor symbol. You may use the logical successor symbol only in an EXAMINE command to examine the entity that logically follows the current entity. You do this by entering an EXAMINE command without an operand and then pressing RETURN.

The logical successor of an entity may not be its physical successor. That is, the logical successor may not occupy the region of physical storage directly following that of the current entity. Such is the case, for example, when the current entity refers to the cell of a disconnected array A(2) whose logical successor A(3) is not stored physically adjacent to A(2) in memory. On the other hand, the logical successor of an entity may also be its physical successor. For instance, in a connected array, A(3) is both the logical and physical successor of A(2).

In some cases the current entity may not have a logical successor. For instance, the last cell of an array has no logical successor, and hence its physical successor is logically unrelated to it.

As with both the current entity and logical predecessor, the debugger uses type and symbolic information associated with the logical successor.

The following example demonstrates how the logical successor symbol may be used to examine succeeding cells in an array:

```
DBG> DEPOSIT CHAR(1) = '1234567890'

DBG> DEPOSIT CHAR(2) = 'ABCDEFGHIJ'

DBG> DEPOSIT CHAR(3) = 'abcdefghij'

DBG> EXAMINE CHAR(1)               !Sets current entity symbol.
MOD\CHAR(1): 1234567890

DBG> EXAMINE RET                   !Logical successor is
MOD\CHAR(2): ABCDEFGHIJ            !CHAR(2).

DBG> EXAMINE RET                   !Next logical successor is
MOD\CHAR(3): abcdefghij            !CHAR(3).
```

3.3 ADDRESS EXPRESSIONS

An address expression specifies a (possibly) typed program location. It may consist of a single operand (a simple address) or of many operands together with operators and delimiters.

The debugger evaluates an address expression according to its own rules, which are similar to those used to evaluate an expression in a programming language. Both an expression and an address expression may include operators and delimiters, and both are evaluated according to rules of precedence. While the result of the evaluation of an expression is a value, the result of the evaluation of an address expression is a 32-bit longword integer that represents a program location.

In an address expression, an operand may be one of the following:

- A simple address
- A unary operator with an operand
- A binary operator with two operands
- An address expression surrounded by parentheses

3.3.1 Operands

Operands in address expressions may be simple addresses, as defined in Section 3.2, or subexpressions, where a subexpression is an expression within an expression. The operands in a subexpression may be simple addresses or other subexpressions.

In the following example, the simple address %LINE 40 is used as an operand in an address expression that also contains the literal 2 as an operand, a multiplication operator, and delimiters:

```
DBG> EXAMINE (%LINE 40)*2
```

This command displays the value at the program location whose address is twice that of %LINE 40.

In the next example, the address expression (X+4)*2 consists of two operands: the subexpression (X+4) and the literal 2.

```
DBG>EVALUATE/ADDRESS (X+4)*2
```

This command calculates the virtual memory address denoted by the address expression (X+4)*2.

In the next example, the address expression ((X-8)*2)+4 contains two subexpressions. The larger subexpression ((X-8)*2) is delimited so as to be one operand for the addition operator, the other operand being the literal 4. The smaller subexpression (X-8) is contained within the larger subexpression and is one of the operands for the multiplication operator, the other operand being the literal 2.

```
DBG>DEPOSIT  ((X-8)*2)+4 = Q
```

This command deposits the value Q in the location represented by the address expression ((X-8)*2)+4.

3.3.2 Operators

An operator with one operand is called a unary operator; an operator with two operands is called a binary operator.

The following is a list of legal operators:

- Plus sign (+): As a unary operator, the plus sign indicates the unchanged value of its operand. As a binary operator, the plus sign adds the preceding operand and succeeding operand together.

- Minus sign (-): As a unary operator, the minus sign indicates the negation of the value of its operand. As a binary operator, the minus sign subtracts the succeeding operand from the preceding operand.

- Multiplication sign (*): A binary operator, the multiplication sign multiplies the preceding operand by the succeeding operand.

- Division sign (/): A binary operator, the division sign divides the preceding operand by the succeeding operand.

- "Contents of" operator (@) or (.): As unary operators, the "at sign" (@) and the period (.) function as "contents of" operators. The "contents of" operator causes its operand to be interpreted as a virtual address and thus requests the "contents of" (or value residing at) that address. Note that these operators are not supported in all languages.

  For example, assume that the value of pointer variable PTR is 7FF00000 hexadecimal, the virtual address of an entity that you want to examine. Assume further that the value of this entity is 3FF00000 hexadecimal. The following command demonstrates how the "contents of" operator is used to examine the entity:

  ```
  DBG>EXAMINE/LONG .PTR
  7FF00000:  3FF00000
  ```

- "Binary shift" operator (@): In VAX-11 MACRO, the "at sign" can be used as a binary operator, in which case it functions to shift the preceding operand the number of bits indicated by the following numeral. The direction of shift is determined by the sign value of the following numeral: positive means shift left; negative, shift right.

NOTE

The debugger interprets the "at sign" (@) according to context: if it is used as a unary operator, it is the "contents of" operator; if used as a binary operator, it is the "binary shift" operator. Similarly, when the period (.) is used as a unary operator, the debugger interprets it as the "contents of" operator; otherwise, the debugger interprets the period as the current entity symbol.

3.3.3 Precedence

Rules of precedence determine the sequence in which the operations of an expression are carried out. Although the debugger evaluates expressions according to its own rules of precedence, which are language-independent, its rules are identical to those of most programming languages.

The order in which the operations within an address expression are carried out is determined by the following three factors, listed in decreasing order of precedence (first listed have higher precedence):

1. The use of delimiters (usually parentheses or brackets) to group operands with particular operators
2. The assignment of relative priority to each operator
3. Left-to-right priority of operators

The following are the legal debugger operators, listed in decreasing order of precedence:

1. Unary operators ((.), (@), (+), (-))
2. Multiplication and division operators ((*), (/))
3. Addition and subtraction operators ((+), (-))

For example, in the evaluation of the address expression:

```
5-(T+5)/4
```

the debugger first adds the operands within parentheses, then divides the result by 4, then subtracts the result from 5.

3.3.4 The EVALUATE/ADDRESS Command

The debugger interprets the operand of an EVALUATE/ADDRESS command as a language-independent address expression, evaluates the address expression using its own rules for expression evaluation (which are similar to those of most languages), and displays the value of the address expression as a virtual address.

The format of the EVALUATE/ADDRESS command is:

```
EVALUATE/ADDRESS address-expression [,address-expression...]
```

If you specify a radix mode command qualifier, the debugger interprets integer literals in the address expression in that radix and displays the value of the address expression as a virtual address in that radix.

In some languages, you can evaluate more than one address expression in a single EVALUATE/ADDRESS command by separating address expressions with a comma.

Any address expression, as discussed in this chapter, may be used with the EVALUATE/ADDRESS command.

The EVALUATE/ADDRESS command is useful in determining the virtual addresses of symbols and of expressions that either contain or do not contain symbols. For example:

```
DBG> EVALUATE/ADDRESS RADIUS                 !For determining address of
1024                                         !a symbol.

DBG> EVALUATE/ADDRESS 1024 + 40              !For performing address
1064                                         !arithmetic.

DBG> EVALUATE/ADDRESS (RADIUS + 4)/2
514
```

CHAPTER 4

EXAMINING AND DEPOSITING DATA

This chapter explains how to use the EXAMINE, DEPOSIT, and EVALUATE commands, and how to control the modes and types used by the debugger. The first section of this chapter discusses the modes available to you. For information on types, see Section 3.1.

4.1 MODES

Modes influence the debugger's interpretation of information you enter in debugger commands, as well as the debugger's display of the results of command execution.

There are two classes of mode:

- Radix mode, which influences the interpretation of numeric literals and, under certain circumstances, the display of data
- Symbolic or nonsymbolic mode, which influences the debugger's display of information under certain conditions

You establish modes by issuing the SET MODE command in the following format:

```
SET MODE mode-keyword [,mode-keyword]
```

Radix mode keywords are DECIMAL, HEXADECIMAL, and OCTAL. Symbolic or nonsymbolic mode keywords are SYMBOL and NOSYMBOL, respectively.

If you do not specify modes by issuing the SET MODE command, the debugger uses default modes that are associated with the current language.

To display the current modes, issue the command:

```
DBG> SHOW MODE
```

To cancel modes established by the SET MODE command, issue the command:

```
DBG> CANCEL MODE
```

Following execution of the CANCEL MODE command, default modes are in effect.

Mode keywords may also be used as command qualifiers in other debugger commands. The following subsections provide a complete description of the use of these mode keywords, both as parameters in the SET MODE command and as command qualifiers in other debugger commands.

4.1.1 Radix Modes

You can specify that the debugger interpret numeric literals and, under certain circumstances, display data in any one of the three following radix modes: DECIMAL, HEXADECIMAL, and OCTAL.

Each language has a default radix mode. Unless you change the default radix mode by using the SET MODE command, the current language's default radix mode is in effect.

For certain commands, you can override the current radix mode for the duration of a command by using a radix mode command qualifier. For example, if you have set the default radix mode to decimal but wish to have the value of an entity displayed in octal, you specify that command qualifier as follows:

```
DBG>  EXAMINE/OCTAL ENTITY
```

When the debugger evaluates a source-language expression or an address expression (in the EVALUATE or EVALUATE/ADDRESS command, respectively), it interprets numeric literals within that expression in the current radix mode unless a radix mode command qualifier is specified. In that case, the debugger interprets numeric literals in the mode specified by the command qualifier.

When the debugger evaluates an address expression (in the EVALUATE/ADDRESS command) it displays the result in the current radix mode unless a radix mode command qualifier is specified. In that case, the debugger displays the result in the radix mode specified by the command qualifier.

In some languages, when the debugger evaluates a source-language expression, it displays the result in the current radix mode unless a radix mode command qualifier is specified. In that case, the debugger displays the result in the mode specified by the command qualifier. In other languages, the debugger may ignore radix qualifiers and display the result of an EVALUATE command in a radix determined by the source language.

Note that when you enter a hexadecimal value that begins with a letter, you must prefix that value with a zero (0); otherwise the debugger attempts to interpret the entry as a symbol.

The following example demonstrates how radix modes are used in EXAMINE, EVALUATE, and EVALUATE/ADDRESS commands:

```
DBG>  SET LANGUAGE PASCAL              !Set language to PASCAL.

DBG>  SHOW MODE                        !Display default modes.
modes: symbolic, decimal

DBG>  SHOW TYPE                        !Display default type.
type: long integer

DBG>  EXAMINE %LINE 15                    !Default type of %LINE is
TOY\%LINE 15: MOVL  #1,B^44(R11)          !instruction in PASCAL.
                                          !Instruction operands are
                                          !displayed in decimal.

DBG>  E/OCTAL/NOSYMBOL                 !Nonsymbolic display of
00000002013:  MOVL  #001,B^054(R11)    !%LINE 15. Instruction
                                       !operands are in octal
                                       !radix.
```

```
DBG>E/HEX/NOSYMBOL                     !Nonsymbolic display of
00000400:  MOVL  #01,B^2C(R11)         !%LINE 15. Instruction
                                       !operands are in hexadecimal
                                       !radix.

DBG>SET LANGUAGE FORTRAN               !Set language to FORTRAN.

DBG>SHOW LANGUAGE                      !Display current language.
language:  FORTRAN

DBG>E/NOSYMBOL PSL                     !Nonsymbolic display of PSL
PSL:  62914592                         !in decimal radix.

DBG>E/OCTAL/NOSYMBOL PSL               !Nonsymbolic display of PSL
PSL:  00360000040                      !in octal radix.

DBG>SET MODE OCTAL                     !Set default radix mode to
                                       !octal.

DBG>SHOW MODE                          !Display current default
modes:  symbolic, octal                !modes.

DBG>EVALUATE/ADDRESS %LINE 15          !Display address of %LINE 15
00000002013                            !in octal (default) radix.

DBG>EVALUATE/ADDRESS/DECIMAL .         !Display address of %LINE 15
1035                                   !in decimal radix.

DBG>EVALUATE/ADDRESS/HEX .             !Display address of %LINE 15
00000408                               !in hexadecimal radix.

DBG>EVALUATE/ADDRESS %LINE 15 - 10 !Perform address arithmetic
00000002003                            !in default octal radix.

DBG>EVALUATE 10 + 8                                  !In octal
%DEBUG-W-INVNUMBER, invalid numeric string '8'       !radix, the
                                                     !character "8"
                                                     !is not valid.

DBG>EVALUATE 7 + 1                     !Use the debugger as a
00000000010                            !calculator. Arithmetic
                                       !performed in octal radix.

DBG>EVALUATE/HEX 7 + 1                 !Arithmetic performed in
00000008                               !hexadecimal radix.

DBG>SET MODE HEX                       !Set default radix mode
                                       !to hexadecimal.

DBG>EVALUATE 9 + 2                     !Arithmetic calculations
0000000B                               !performed in hexadecimal
                                       !radix.

DBG>EVALUATE/ADDRESS %LINE 15 - B
%DEBUG-W-NOSYMBOL, symbol 'B' is not in the symbol table

DBG>EVALUATE/ADDRESS %LINE 15 - 0B !Prefix the 'B' by a '0'
00000400                               !(zero) and the address
                                       !arithmetic is performed.
```

4.1.2 Symbolic and Nonsymbolic Modes

If symbolic mode is set, the debugger displays information in its symbolic representation (if there is such a representation). If nonsymbolic mode is set, the debugger displays symbolic information in its numeric equivalent in the current radix mode. Note that symbolic mode only affects the debugger's display; you can enter both symbolic or nonsymbolic values regardless of mode settings.

The symbolic and nonsymbolic modes influence the debugger's display of information in the three following situations:

1. **In the display of an address expression.** In symbolic mode, the debugger displays the location denoted by an address expression in its symbolic representation (if there is such a representation). In nonsymbolic mode, the debugger displays the location denoted by an address expression as a virtual address, whether or not that virtual address is represented by a symbol.

   The following example demonstrates the effects of symbolic and nonsymbolic modes in the display of address expressions:

```
DBG> SHOW MODE                     !Display current default
modes: symbolic, decimal           !modes.

DBG> EXAMINE J                     !Address expression J is
TOY\J:  44444                      !displayed in full symbolic
                                   !form (that is, as TOY\J).

DBG> EXAMINE/NOSYMBOL J            !Specify nonsymbolic mode.
712:  44444                        !Value of J is displayed as
                                   !virtual address 712.

DBG> SET MODE NOSYMBOL             !Set default mode to
                                   !nonsymbolic.

DBG> SHOW MODE                     !Display new default mode.
modes: nosymbolic, decimal

DBG> EXAMINE 712                   !Examine virtual address.
712:  44444

DBG> EXAMINE J                     !Mode does not affect the
712:  44444                        !interpretation of values you
                                   !enter. J is an acceptable
                                   !parameter in the command,
                                   !even though default mode
                                   !is nonsymbolic.

DBG> EXAMINE/SYMBOL J              !Specify symbolic display.
TOY\J:  44444
```

2. **In the display of the operands in an instruction.** In symbolic mode, the debugger attempts to translate an instruction operand to its symbolic equivalent. In nonsymbolic mode, any operand in an instruction is displayed numerically, whether or not it has a symbolic representation.

   The following example demonstrates the effects of symbolic and nonsymbolic modes in the display of instruction operands:

```
DBG> SHOW MODE                     !Display default modes.
modes:  symbolic, decimal
```

```
DBG> EXAMINE/INSTRUCTION OUT                    !Display OUT as an instruction
DRAW\OUT:  CMPB  B^DRAW\COL.#0                  !using the default symbolic
                                                !mode. Note that the
                                                !instruction operand DRAW\COL
                                                !is expressed symbolically.

DBG> EXAMINE/INSTRUCTION/NOSYMBOL OUT           !Specify nonsymbolic mode.
3678:  CMPB    B^3599,#0                        !Note that the instruction
                                                !operand DRAW\COL is now
                                                !expressed numerically.

DBG> EXAMINE/INSTRUCTION/NOSYMBOL 3678          !Specify nonsymbolic mode.
3678:  CMPB  B^3599,#0                          !Note that the instruction
                                                !operand DRAW\COL is now
                                                !expressed numerically.

DBG> EXAMINE/INSTRUCTION 3678                   !In symbolic mode, you may
DRAW\OUT:  CMPB    B^DRAW\COL.#0                !enter information either
                                                !symbolically or nonsymbol-
                                                !ically. The nonsymbolic
                                                !location of OUT is 3678.
                                                !Since default mode is sym-
                                                !bolic, display is symbolic.

DBG> EXAMINE/INSTRUCTION/NOSYMBOL 3678          !In nonsymbolic mode, you
3678:  CMPB  B^3599,#0                          !may enter information
                                                !either symbolically or
                                                !nonsymbolically. Since
                                                !NOSYMBOL command override
                                                !is used, display is
                                                !nonsymbolic.

                                                !Note that the instruction
                                                !operand DRAW\COL is
                                                !expressed
                                                !nonsymbolically.
```

3. **In the display of the Processor Status Longword (PSL).** In symbolic mode, execution of the command:

   ```
   DBG> EXAMINE PSL
   ```

   results in a formatted display of the contents of the PSL. In nonsymbolic mode, the PSL is displayed in numeric form.

   The following example demonstrates the effect of symbolic and nonsymbolic modes in the display of the PSL:

```
DBG> SHOW MODE                                  !Display default modes.
modes: symbolic, decimal

DBG> EXAMINE PSL
PSL:   CMP  TP  FPD  IS  CURMOD  PRVMOD  IPL  DV  FU  IV  TN  Z  V  C
        0    0    0   0   USER    USER    00   0   0   1  0   0  0  0

                                                !Symbolic mode gives
                                                !formatted display of PSL.

DBG> EXAMINE/NOSYMBOL PSL                       !Nonsymbolic mode gives
PSL:   62914592                                 !unformatted display of PSL.
```

See Section 7.4.3.1 for more information about the PSL.

4.2 THE EXAMINE COMMAND

You use the EXAMINE command in the following format to display the value of one or more program entities:

```
EXAMINE [/qualifier[/qualifier...]] -
        [address-expression[:address-expression]-
        [,address-expression[:address-expression]...]]
```

If you specify a single address-expression parameter in the EXAMINE command, the debugger displays the value of the entity at the location denoted by the address expression in the type associated with that location.

You can examine more than one entity (a list) in a single EXAMINE command by entering more than one address expression and separating each with a comma (,).

In some languages, you can examine a range of entities in a single EXAMINE command by entering the address expression that denotes the first entity in the range, a colon (:), and the address expression that denotes the last entity in the range. A range is a contiguous sequence of program entities.

In some languages, you can examine a list of ranges of entities in a single EXAMINE command by separating each range with a comma.

If you specify a type command qualifier in the EXAMINE command, the debugger displays the entity or entities in that type.

If you specify a radix mode command qualifier in the EXAMINE command, the debugger interprets numeric literals in the address expression(s) in the specified radix and displays the examined entity or entities in the specified radix.

If you specify the /SYMBOL mode command qualifier in the EXAMINE command, the debugger displays address expression(s) and instruction operand(s) symbolically (if possible). If you specify the /NOSYMBOL mode command qualifier, symbolic information is displayed in its numeric equivalent.

When you examine the PSL in SYMBOL mode, the debugger displays its contents in a formatted arrangement; otherwise, the debugger displays its contents as a numeric literal in the current radix mode.

Address expressions may take any of the forms discussed in Chapter 3. See Section 3.1.1 for a complete description of how the debugger associates types with address expressions.

The following sections discuss the use of command qualifiers in the EXAMINE command, the examining of lists, the examining of ranges, and the examining of successive entities.

4.2.1 Command Qualifiers

If you specify a type command qualifier in the EXAMINE command, the entity specified by the address expression is displayed in that type. The type command qualifiers are /BYTE, /WORD, /LONGWORD, /QUADWORD, /OCTAWORD, /FLOAT, /D_FLOAT, /G_FLOAT, /H_FLOAT, /ASCII:n, and /INSTRUCTION.

You may also specify the radix mode command qualifiers (/HEXADECIMAL, /DECIMAL, and /OCTAL) and/or the symbolic and nonsymbolic mode command qualifiers (/SYMBOL, /NOSYMBOL). See Section 4.1 for further information on the effects of specifying these mode command qualifiers.

It is possible to specify a type qualifier, a radix mode qualifier, and a symbolic/nonsymbolic mode qualifier in a single EXAMINE command.

The following example demonstrates the use of these mode and type command qualifiers in the EXAMINE command:

```
DBG> SET LANGUAGE FORTRAN               !Set language to FORTRAN.

DBG> SHOW MODE                          !Display default modes.
modes: symbolic, decimal

DBG> SHOW TYPE                          !Display default type.
type: long integer

DBG> EXAMINE %LINE 15                   !Display %LINE 15 in default
TOY\%LINE 15 :  749404624               !modes and type.

DBG> EXAMINE/NOSYMBOL .                 !Nonsymbolic display of
1035:  749404624                        !%LINE 15.

DBG> EXAMINE/BYTE .                     !Type is byte integer.
TOY\%LINE 15 :  -48

DBG> EXAMINE/WORD .                     !Type is word integer.
TOY\%LINE 15 :  464

DBG> EXAMINE/LONG .                     !Type is long integer.
TOY\%LINE 15 :  749404624

DBG> E/QUAD .                           !Type is quadword integer.
TOY%LINE 15 :  +0130653502894178768

DBG> E/OCTAWORD .                       !Type is octaword integer.
TOY%LINE 15 :  -112752446431792802470086860782410464 81

DBG> E/FLOAT .                          !Type is F_floating.
TOY%LINE 15 :    1.9117807E-38

DBG> E/D FLOAT .                        !Type if D_floating.
TOY%LINE 15 :     1.9117807293306393E-38

DBG> E/G FLOAT .                        !Type is G_floating.
TOY%LINE 15 :     1.509506018605227E-300

DBG> E/H FLOAT .                        !Type is H_floating.
TOY%LINE 15 :   2.351187242166315296772081991217048E-4793

DBG> EXAMINE/INSTRUCTION .                 !Type is VAX-11 instruction.
TOY\%LINE 15 :  MOVL    #1,B^44(R11)

DBG> EXAMINE/ASCII .                    !Type is ASCII string.
TOY\%LINE 15 :

DBG> EXAMINE/OCTAL .                    !Radix mode is octal.
TOY\%LINE 15 :  05452600720             !Default type is longword.
```

```
DBG> EXAMINE/DECIMAL                 !Radix mode is decimal.
TOY\%LINE 15 : 748404624             !Default type is longword.

DBG> EXAMINE/HEX .                   !Radix mode is hexadecimal.
TOY\%LINE 15 : 2CAB01D0              !Default type is longword.

DBG> EXAMINE/BYTE/HEX .              !Type is byte. Radix mode
TOY\%LINE 15 : 0D0                   !is hexadecimal.

DBG> EXAMINE/BYTE/OCT .              !Type is byte. Radix mode
TOY\%LINE 15 : 320                   !is octal.

DBG> EXAMINE/NOSYMBOL/BYTE/HEX       !Nonsymbolic display of one
0000040B:  0D0                       !byte in hexadecimal radix.

DBG> EXAMINE/INSTRUCTION/BYTE .      !When two types are specified,
TOY\%LINE 15 : -48                   !the last overrides the first.
```

4.2.2 Examining Lists

You can examine any number of program locations using a single EXAMINE command by separating parameters in the command string with a comma. Program locations may be listed randomly. The following is the format used:

```
EXAMINE  address-expression, address-expression, ...
```

The following example shows how you examine a list of three variables by means of a single EXAMINE command:

```
DBG> EXAMINE I,K,R
TOY\I:  0
TOY\K:  0
TOY\R:  0.0000000E+00
```

4.2.3 Examining Ranges

In some languages, you can examine a range of successive program locations using a single EXAMINE command by specifying the first and last program locations in the range, in the following format:

```
EXAMINE address-expression1 :  address-expression2
```

Address-expression1 must have a smaller virtual memory address than address-expression2; otherwise the debugger issues an error message.

Note that one or more blank spaces or tabs between the colon (:) and address-expressions are optional.

As a result of this command, the debugger displays the entity specified by address-expression1, the logical successor of address-expression1, the next logical successor, and so on, until it displays the entity specified by address-expression2.

The debugger associates a type with address-expression1 according to the rules described in Section 3.1.1.

Note that you can use a single EXAMINE command to examine more than one range of program locations (in other words, a list of ranges) by separating ranges with a comma. The following is the format for specifying a list of ranges:

```
EXAMINE address-expression1 :  address-expression2, -
    address-expression3 :  address-expression4, ...
```

Note that the hyphen (-), which is the line continuation character, is used to permit continued entry of the command string on a new line.

The following example demonstrates the examination of ranges of program locations:

```
DBG> EXAMINE R:I                                      !R has a larger
%DEBUG-W-EXARANGE, invalid range of addresses         !address than I.

DBG> EXAMINE I:R                          !Modes are the defaults
TOY\I:  555                               !symbolic and decimal.
TOY\J:  0                                 !Logical successor of I is J.
TOY\R:  0.0000000E+00                     !Logical successor of J is R.
                                          !I and J have the type
                                          !longword. R has the floating
                                          !type.

DBG> EXAMINE/BYTE I:R                     !Byte is the override type.
TOY\I:  43                                !The range of locations is
TOY\I+1:  2                               !displayed as a series of
TOY\I+2:  0                               !bytes.
TOY\I+3:  0
TOY\J:  0
TOY\J+1:  0
TOY\J+2:  0
TOY\J+3:  0
TOY\R:  0

DBG> EXAMINE/BYTE/NOSYMBOL I:R            !Nonsymbolic mode causes
708:  43                                  !the location of each byte to
709:  2                                   !be displayed as a virtual
710:  0                                   !address, not as a symbol.
711:  0
712:  0
713:  0
714:  0
715:  0
716:  0
```

4.2.4 Examining Successive Entities

You can examine the value of successive entities by using the logical successor symbol RETURN. The debugger uses the type and mode associated with the current entity to interpret logical successors.

The following example demonstrates how to examine successive entities using the logical successor symbol RETURN:

```
DBG> EXAMINE I                            !Default type is longword.
TOY\I:  555
```

```
DBG> EXAMINE (RET)                          !Display logical successor
TOY\J:  0                                   !of I.

DBG> EXAMINE (RET)                          !Display next logical
TOY\R:  0.0000000E+00                       !successor of I.

DBG> EXAMINE (RET)                          !Display next logical
TOY\K:  0                                   !successor of I.

DBG> EXAMINE VECTOR(1)                      !Display value of VECTOR(1).
TOY\VECTOR(1):  0.0000000E+00

DBG> E (RET)                                !Display logical successor.
TOY\VECTOR(2):  0.0000000E+00
```

4.3 THE DEPOSIT COMMAND

You use the DEPOSIT command to deposit values in program locations. In the following command format, the expression designates the value to be deposited, and the address expression designates the program location into which the value is to be deposited:

```
DEPOSIT [/qualifier] address-expression = expression [,expression...]
```

The expression is evaluated in the syntax of the source language and in the specified (or default) radix mode to yield a value. The address expression is evaluated to yield a program location with an associated type.

When the debugger executes the DEPOSIT command, it converts the value of the expression to the type associated with the address expression and deposits the value at the location designated by the address expression.

In some languages, you can deposit more than one expression with a single DEPOSIT command by listing the expressions to be deposited on the right side of the equal sign, separated by commas. The debugger deposits the value of the first expression into the location identified by the address expression (the current entity). The debugger then deposits the value of the next expression into the logical successor of the current entity and the values of remaining expressions into subsequent logical successors.

You may specify a type command qualifier in the DEPOSIT command to associate a type with the program location denoted by the address expression. If, in addition, you specify more than one expression, the debugger associates the specified type with one or more program locations (depending on the number of expressions) that are logical successors of the specified address expression.

If a particular language does not support a debugger type (such as one of the floating point, quadword integer, or octaword integer types), you may specify that type as a command qualifier in the DEPOSIT command only if you enclose the expression(s) to be deposited in quotation marks or apostrophes and only if the expression(s) contain no operators or delimiters. In this case, the debugger uses its own type-conversion rules rather than those of the language. For example,

```
DBG> DEPOSIT/FLOAT X = '123.5'
```

You may also specify a radix command qualifier in the DEPOSIT command to influence the interpretation of integers in both the address expression and the expression(s).

The following subsections discuss the depositing of numeric data, ASCII strings, and VAX-11 instructions, as well as the use of radix and mode qualifiers in the DEPOSIT command.

4.3.1 Depositing Numeric Data

The following example demonstrates how to deposit three integer values into three memory locations using a single DEPOSIT command:

```
DBG> SHOW MODE                         !Display default modes.
modes:  symbolic, decimal
DBG> DEPOSIT J = 333,444,555           !Deposit data in J, R, K.
DBG> EXAMINE .                         !Current entity is location
TOY\K:  555                            !last deposited into.
DBG> EXAMINE J                         !The value 333 is deposited
TOY\J:  333                            !in J.
DBG> EXAMINE (RET)                     !The value 444 is deposited
TOY\R:     444.0000                    !into the logical successor of
                                       !J, which is R. The type
                                       !associated with R is used in
                                       !display, not the default
                                       !type.
DBG> EXAMINE (RET)                     !The value 555 is deposited
TOY\K:  555                            !into the logical successor of
                                       !R, which is K.
```

The next example demonstrates the use of the byte, word, and longword type command qualifiers in the DEPOSIT command:

```
DBG> SHOW TYPE                         !Display default type.
type:  long integer
DBG> EVALU/ADDR .                      !Current location is 724.
724
DBG> DEPOSIT/BYTE . = 1                !Deposit the value 1 into the
                                       !byte of memory whose address
                                       !is 724.
DBG> E .                               !Because the default type is
724:  1280461057                       !long integer, 4 bytes are
                                       !examined.
DBG> E/BYTE .                          !Examine one byte only.
724:  1
DBG> DEPOSIT/WORD . = 2                !Deposit the value 2 into the
                                       !first two bytes (word) of
                                       !the current entity.
DBG> E/WORD .                          !Examine a word of the current
724:  2                                !entity.
```

```
DBG> DEPOSIT/LONG 724 = 9999      !Deposit the value 9999 into
                                  !4 bytes (a longword) begin-
                                  !ning at virtual address 724.

DBG> E/LONG 724                   !Examine 4 bytes (longword)
724:  9999                        !beginning at virtual address
                                  !724.
```

4.3.2 Depositing ASCII Strings

You can deposit an ASCII string into a program location only if that location is denoted by an address expression with an associated ASCII type.

Delimiters may be either apostrophes (') or quotation marks ("); however, the delimiter used must not appear within the string itself.

In all cases, if the length of the string to be deposited exceeds the length of the program location into which it is deposited, the debugger truncates the string to the length of the program location. On the other hand, if the length of the string is less than the length of the program location, the debugger inserts ASCII blanks in the remaining bytes of memory to the right of the last character in the string.

The following sections discuss the depositing of ASCII strings into program locations denoted by address expressions that:

- Have an ASCII type
- Have a non-ASCII type
- Have no type

4.3.2.1 Depositing Into an Address Expression With an ASCII Type - To deposit an ASCII string into a program location denoted by an address expression of ASCII type, issue the DEPOSIT command in the following format:

```
DEPOSIT address-expression = "ASCII string"
```

4.3.2.2 Depositing Into an Address Expression With a Non-ASCII Type - If the program location is denoted by an address expression of a type other than ASCII, you can override that type by using the command SET TYPE/OVERRIDE ASCII:n. This is especially useful when you intend to issue several DEPOSIT commands.

You can also use the /ASCII:n command qualifier with the DEPOSIT command to override any associated type only for the duration of the command. This is especially useful when you are issuing a single DEPOSIT command.

The following command format demonstrates the use of the /ASCII:n command qualifier with the DEPOSIT command:

```
DEPOSIT/ASCII:n address-expression = "ASCII string of length n"
```

Note that the value n used in the command qualifier should normally correspond to the number of ASCII characters to be deposited, that is, to the length of the ASCII string.

4.3.2.3 Depositing Into an Address Expression Without a Type - If the program location is denoted by an address expression that does not have an associated type, you can associate the ASCII type with the address expression by issuing the command SET TYPE ASCII:n. Subsequent to this, you can use the DEPOSIT command format for address expressions of type ASCII as shown in Section 4.3.2.1.

You can also use the /ASCII:n command qualifier with the DEPOSIT command to associate the ASCII type with the address expression that denotes the program location into which the string is to be deposited. The command format to be used in this case is shown in Section 4.3.2.2.

Note that you may also use the SET TYPE/OVERRIDE ASCII:n to associate the type ASCII with an untyped address expression.

The following example demonstrates how to deposit ASCII strings:

```
DBG> SET TYPE ASCII                     !Set default to ASCII.
                                        !When no length is specified,
                                        !the debugger defaults to 4.

DBG> SHOW TYPE                          !Display the default type.
type:  ASCII:  4

DBG> DEPOSIT I = GOOD
%DEBUG-WNOSYMBOL, symbol 'GOOD' is not in the symbol table

                                        !Problem is no string
                                        !delimiters.

DBG> DEPOSIT I = "GOOD"
%DEBUG-W-INVNUMBER, invalid numeric string 'GOOD'

                                        !Debugger is expecting a
                                        !number because I has an
                                        !associated numeric type. You
                                        !must override that type.

DBG> DEPOSIT/ASCII I = "GOOD"           !Use ASCII override type.
                                        !Debugger deposits the string.

DBG> EXAMINE .
TOY\I:  1146048327                      !Display in default type.

DBG> EXAMINE/ASCII .                    !Specify ASCII display.
TOY\I:  GOOD

DBG> SET TYPE/OVERRIDE ASCII            !Specify ASCII override
                                        !type.

DBG> DEPOSIT I = "GOOD","LUCK","BOYS","AND","GIRLS"
%DEBUG-I-STGTRUNC, string truncated

                                        !Deposits a list of
                                        !ASCII strings; truncates
                                        !"girls" to "girl" because
                                        !type is 4-byte ASCII.

DBG> EXAMINE I
TOY\I:  GOOD                            !Display in ASCII override
                                        !type.

DBG> EXAMINE RET                        !Display logical successor.
TOY\J:  LUCK
```

```
DBG> EXAMINE RET                     !Display next logical
TOY\R:  BOYS                         !successor.

DBG> EXAMINE RET                     !Display next logical
TOY\K:  AND                          !successor.

DBG> EXAMINE RET                     !Display next logical
724:  GIRL                           !successor.

DBG> CANCEL TYPE/OVERRIDE            !Cancel ASCII override type.

DBG> EXAMINE I
TOY\I:  1146048327                   !Compiler-generated type
                                     !overrides the default ASCII
                                     !type.
```

4.3.3 Depositing VAX-11 Instructions

To deposit VAX-11 instructions, the following command format may be used:

DEPOSIT/INSTRUCTION address-expression = "VAX-11 instruction"

You must enclose the instruction in either apostrophes (') or quotation marks ("). Either delimiter may be used so long as it does not appear within the instruction string. The first and last delimiter must be the same.

You must tell the debugger that the delimited string is to be interpreted as an instruction and not as an ASCII string. You do this by specifying the INSTRUCTION type as a default type or an override type, depending on the type associated with the address expression into which you are depositing and on the default and override types currently in effect.

If you are depositing an instruction into a location specified by an address expression that does not have an associated type, then you must associate the INSTRUCTION type with that address expression. You may use the SET TYPE INSTRUCTION command to change the default type to INSTRUCTION, or you may use an override type of INSTRUCTION.

If you are depositing an instruction into a location specified by an address expression that has another associated type, then you must override that type by issuing the SET TYPE/OVERRIDE INSTRUCTION command or by specifying the /INSTRUCTION command qualifier in the DEPOSIT command.

Note that if there is another override type in effect, you must specify the INSTRUCTION type as a qualifier in the DEPOSIT command.

Depositing VAX-11 instructions is simple when you are depositing into successive memory locations because you can use the logical successor symbol to locate the address where the next instruction is to be deposited. The following example demonstrates this:

```
DBG> SET TYPE INSTRUCTION            !Set default type to
                                     !instruction.

DBG> DEPOSIT 730 = "MOVB #77, R1"    !Deposit the instruction
                                     !beginning at virtual address
                                     !730.
```

```
DBG> EXAMINE .                         !Examine current entity.
730:  MOVB  #77,R1

DBG> EXAMINE (RET)                     !Make current entity the
734:  HALT                             !logical successor of virtual
                                       !address 730.

DBG> DEPOSIT . = "MOVB #66, R2"        !Deposit next instruction.

DBG> EXAMINE .                         !Examine current entity.
734:  MOVB  #66,R2

DBG> EXAMINE (RET)                     !Make current entity the
738:  HALT                             !logical successor of virtual
                                       !address 734.

DBG> DEPOSIT . = "MOVB #55, R3"        !Deposit next instruction.

DBG> EXAMINE .                         !Examine current entity.
738:  MOVB  #55,$3
```

The replacing of one or more VAX-11 instructions with new ones involves keeping track of the lengths of the instructions, which vary depending on the type of instruction and the number of operands. See Chapter 7 on assembly-level debugging for more information about replacing instructions.

4.3.4 Depositing in Different Radixes

Literals in source-language expressions are interpreted by the debugger in the current radix mode or in the radix specified by a radix mode command qualifier. The following example demonstrates how to deposit literals in the octal, decimal, or hexadecimal radix:

```
DBG> SHOW MODE                         !Display default modes.
modes: symbolic, decimal

DBG> SHOW TYPE                         !Display default type.
type: long integer

DBG> EXAMINE J                         !Display J in default
TOY\J:  234567890                      !modes and type.

DBG> DEPOSIT/OCTAL J = 7777777         !Deposit an octal value.

DBG> EXAMINE .                         !Display in compiler-generated
TOY\J:  2097151                        !type and default radix
                                       !(decimal).

DBG> EXAMINE/OCTAL .                   !Display in compiler-generated
TOY\J:  00007777777                    !type and octal radix.

DBG> DEPOSIT/HEX J = 7777777           !Deposit a hexadecimal
                                       !value.

DBG> EXAMINE .                         !Display in type
TOY\J:  125269879                      !associated with J and in
                                       !default radix (decimal).

DBG> EXAMINE/HEX .                     !Display in hexadecimal.
TOY\J:  07777777
```

4.4 THE EVALUATE COMMAND

The debugger interprets the operand of an EVALUATE command as a source-language expression, evaluates it in the semantics of the source language, and displays its value as a literal in the source language.

The format of the EVALUATE command is:

```
EVALUATE [/qualifier] expression [,expression...]
```

If you specify a radix mode command qualifier, the debugger interprets integer literals in the expression (or expressions) in that radix; and in some languages, the debugger displays the value of the expression (or expressions) in that radix.

If an expression contains symbols with different compiler-generated types, the debugger uses the type-conversion rules of the current language to evaluate the expression.

In some languages, you can evaluate more than one expression in a single EVALUATE command by separating expressions with a comma.

Using the EVALUATE command, you can perform arithmetic calculations that may or may not be related to your program. You may also perform radix conversions by using radix mode command qualifiers.

The following examples demonstrate how to use the EVALUATE command (language is set to FORTRAN):

```
DBG> DEPOSIT R = 5.35E3              !R is of type floating point.

DBG> EVALUATE R                      !Gives the value of R.
   5350.000

DBG> EXAMINE R                       !The EXAMINE command is
TOY\R:  5350.000                     !similar. Note that symbolic
                                     !address of R is displayed.

DBG> EVALUATE R*50                   !Perform arithmetic
   267500.0                          !calculation.

DBG> EVALUATE (R*50)/(540-40)        !Perform arithmetic
   535.0000                          !calculation.

DBG> DEPOSIT I = 22222               !Deposit value in I.

DBG> EVALUATE I                      !Note that I is of type
22222                                !integer.

DBG> EVALUATE R/I                    !Expression is evaluated
   0.2407524                         !in floating point.

DBG> EVALUATE 2.5*35                 !Perform an arithmetic
   87.50000                          !calculation.

DBG> EVALUATE/OCTAL 17/2             !Arithmetic calculation in
00000000007                          !octal.

DBG> EVALUATE/HEX 17/2               !Arithmetic calculation in
0000000B                             !hexadecimal.

DBG> EVALUATE/DEC 17/2               !Arithmetic calculation in
8                                    !decimal.
```

CHAPTER 5

PROGRAM CONTROL

This chapter explains how to start, suspend, and monitor program execution, and how to display the current program status. It also discusses how to debug exit handlers.

5.1 STARTING PROGRAM EXECUTION

To begin program execution at debugger start up or to continue program execution after interruption, issue the STEP, GO, or CALL command. The following subsections discuss each of these in turn.

5.1.1 The STEP Command

By means of the STEP command, you can control the execution of your program. When you issue a STEP command, the debugger responds as follows:

1. It reports the next line or instruction to be executed.
2. It executes one or more instructions.
3. It reports the line or instruction that follows the last instruction executed.
4. It reports the source line corresponding to the line or instruction that follows the last instruction executed only if the SOURCE parameter is in effect by virtue of STEP/SOURCE or SET STEP SOURCE and source lines are available.
5. It issues its prompt.

The format of the STEP command is:

```
STEP [/qualifier...] [integer]
```

The decimal integer is an optional parameter that indicates how many lines or instructions are to be executed. If the decimal integer is omitted, the debugger uses a default value of 1.

The STEP command qualifiers specify whether the debugger steps by line or by instruction, whether the debugger steps "into" or "over" called routines in the user program space (user-written routines), whether the debugger steps "into" or "over" called routines in system space, and whether the debugger displays lines of source code as it steps. (System space is defined as those virtual addresses in memory greater than 80000000 hexadecimal.)

If STEP command qualifiers are not specified with the STEP command, the debugger uses STEP parameters established by the SET STEP command (see Section 5.1.1.1), or STEP parameters associated by default with the current language.

The following list describes the effects of the STEP command qualifiers:

- /LINE -- A single step causes execution of all the user code between the location where execution was last suspended and the next line boundary.

- /INSTRUCTION -- A single step causes execution of a single instruction at the location where execution was last suspended.

- /INTO -- The debugger steps "into" called routines in the user program space (and "into" called routines in system space if /SYSTEM is also specified). That is, the debugger does not differentiate between instructions (or lines) within a called routine and those outside of the routine as it executes steps. Thus, the execution of a STEP command may result in suspension of execution within the called routine, in which case subsequent step commands will execute the instructions (or lines) within that routine.

- /OVER --The debugger steps "over" called routines in the user program space and in system space. That is, the debugger considers any instructions executed as the result of a CALL instruction, up to and including the corresponding RETURN instruction, to be part of a single step.

- /SYSTEM -- When this STEP qualifier is in effect, the debugger steps "into" called routines in system space, provided that /INTO is also specified. That is, the debugger, in its execution of steps, does not differentiate between instructions (or lines) within a called routine in system space and those outside of system space. Thus, execution of a STEP command may result in suspension of execution in system space.

- /NOSYSTEM -- The debugger steps "over" called routines in system space. That is, the debugger considers any instructions executed as the result of a CALL instruction to a routine in system space, up to and including the corresponding RETURN instruction, to be part of a single step.

- /SOURCE -- In some languages, when this STEP qualifier is in effect, the debugger displays the line of source code that corresponds to the instruction(s) being executed. See Chapter 8 for more information.

- /NOSOURCE -- In some languages, when this STEP qualifier is in effect, the debugger does not display the line of source code that corresponds to the instruction(s) being executed. See Chapter 8 for more information.

Note that when the step parameter OVER is in effect, the debugger steps "over" called routines both in the user program space and in system space. Consequently, to step "into" called routines in system space, you must specify the parameters INTO and SYSTEM.

When a qualifier is specified with the STEP command, it overrides, for the duration of that command, the corresponding STEP parameter established by the SET STEP command or the corresponding default STEP parameter associated with the current language.

5.1.1.1 The SET STEP and SHOW STEP Commands

The SET STEP command sets the default step parameters; that is, it establishes the step parameters that the debugger uses whenever a STEP command is issued without a STEP command qualifier.

When a step parameter is specified as a qualifier in the STEP command, it overrides, for the duration of the command, any step parameters established by the SET STEP command or any step parameters associated with the current language.

As described in Section 5.1.1, step parameters may be any of the following: LINE, INSTRUCTION, INTO, OVER, SYSTEM, NOSYSTEM, SOURCE, and NOSOURCE. Note however that a step parameter is not prefixed with a slash (/) unless it is used as a STEP command qualifier.

The following is the format of the SET STEP command:

```
SET STEP step-parameter [,step-parameter...]
```

Note that default step parameters are associated with each language. Thus, when you change languages by using the SET LANGUAGE command, the default step parameters may also change.

To display the current step parameters, issue the command:

```
DBG> SHOW STEP
```

5.1.2 The GO Command

You can use the GO command:

- To begin execution of your program at debugger start up, in which case execution begins from the transfer address
- To resume execution at the instruction following the last instruction executed, in which case you do not specify an address expression as a parameter in the GO command
- To resume execution at a specified program location, in which case you specify the corresponding address expression as a parameter in the GO command

Unlike the STEP command, which suspends execution after a specified number of instructions or lines have been executed, the GO command will cause your program to continue executing until one of the following occurs:

- The program terminates.
- A breakpoint is reached.
- A watchpoint is activated.
- An exception occurs.
- The program is interrupted by CTRL/Y or CTRL/C.

The following is the format of the GO command:

```
GO [address-expression]
```

If an address expression is specified as a parameter in the GO command, the debugger begins program execution at the location specified by the address expression.

Using an address expression as a parameter in the GO command may be helpful in situations where you want to re-execute sections of your program but do not want to start all over again. Note however that when you specify an address-expression parameter in the GO command, the program state at the location denoted by the address expression must be identical to the program state at the time you issue the GO command; otherwise, results may be unpredictable.

See Section 5.2.1.2 for examples of how to use the GO command.

5.1.3 The CALL Command

The CALL command is useful in debugging routines in your program.

When you issue a CALL command, the debugger takes the following action:

1. Saves the current values of the general registers
2. Constructs an argument list
3. Executes a call to the routine specified in the command and passes any arguments
4. Executes the routine
5. Displays the value returned by the routine in R0
6. Restores the values of the general registers to the values they had just previous to the CALL command
7. Issues its prompt

The debugger executes a routine called by the CALL command whether or not your program actually includes a call to that routine, so long as the routine was linked with your program.

You can also debug unrelated routines by linking them with a dummy main module that has a transfer address and then using the CALL command to execute them.

The following is the format of the CALL command:

```
CALL routine-name [ ( argument1[,argument2...] ) ]
```

Note that any arguments must be enclosed in parentheses.

5.2 SUSPENDING PROGRAM EXECUTION

The following sections discuss how to suspend program execution by setting breakpoints, exception breakpoints, and watchpoints.

5.2.1 Breakpoints

A breakpoint is a program location at which the debugger:

1. Suspends program execution
2. Displays the name or the virtual address of the location at which execution has been suspended
3. Executes commands in a DO command sequence if one was specified in the SET BREAK command
4. Issues its prompt

To set a breakpoint, issue the SET BREAK command in the following format:

```
SET BREAK [/qualifier] address-expression [DO (command [;command...])]
```

Note that the debugger suspends program execution at the first byte of the location denoted by the address expression in the SET BREAK command and therefore does not execute the instruction that begins at that location.

If you specify the /AFTER:n command qualifier, the debugger takes break action at the nth activation of the specified location. See Section 5.2.1.1.

If you specify a DO command sequence, the debugger executes commands in the sequence when the breakpoint is activated. See Section 5.2.1.2.

If you use a virtual memory address or an address expression whose value is not a symbolic location as a parameter in the SET BREAK command, you should check that an instruction actually begins at the byte of memory so indicated. If an instruction does not begin at this byte, a run-time error may occur when an instruction including that byte is executed. Note that when the breakpoint is set, the debugger does not verify that the location specified marks the beginning of an instruction.

When a routine name is the parameter in a SET BREAK command, the debugger notes this fact when the breakpoint is activated by issuing the following message:

```
routine break at routine name
```

Routine breakpoints are special cases because the breakpoint is actually set at the memory address two bytes greater than the memory address of the routine name itself (called the entry point). This is done so that the breakpoint is not set on the 2-byte entry mask of the routine, but is set instead at the first instruction in the routine, which begins directly following the entry mask.

Note that the command:

```
DBG> EXAMINE routine-name
```

does not skip over the routine entry mask. Therefore, to examine the first instruction in the routine, you must issue the command:

```
DBG> EXAMINE routine-name + 2
```

To see what breakpoints are in effect, issue the command:

```
DBG> SHOW BREAK
```

Once set, a breakpoint remains active for the duration of the debugging session unless you cancel it with the CANCEL BREAK command or set another breakpoint, watchpoint, or tracepoint at that program location. In that case the old breakpoint specification is overwritten.

To cancel one or more breakpoints, issue the CANCEL BREAK command in the following format:

```
CANCEL BREAK [/qualifier] [address-expression]
```

If you specify an address-expression parameter, the breakpoint at the location denoted by the address expression is canceled. In this case, you cannot also specify a command qualifier.

If you specify the /ALL command qualifier, all breakpoints are canceled. In this case, you cannot also specify an address-expression parameter.

The following example shows how breakpoints may be set, canceled, and overwritten:

```
DBG> SET BREAK %LINE 15                   !Set breakpoint.

DBG> SHOW BREAK                           !Show breakpoint.
breakpoint at TOY\%LINE 15

DBG> CANCEL BREAK %LINE 15                !Cancel breakpoint.

DBG> SHOW BREAK
%DEBUG-I-NOBREAKS, no breakpoints are set

DBG> SET BREAK %LINE 15                   !Set breakpoint.

DBG> SET WATCH %LINE 15                   !Set watchpoint at same
                                          !location.

DBG> SHOW BREAK                           !Breakpoint is overwritten.
%DEBUG-I-NOBREAKS, no breakpoints are set

DBG> SHOW WATCH                           !Watchpoint set.
watchpoint at TOY\%LINE 15 for 4.  bytes.

DBG> SET BREAK %LINE 15                   !Set breakpoint.

DBG> SET BREAK/AFTER:5 %LINE 15           !Set modified breakpoint
                                          !at same location.

DBG> SHOW BREAK                              !Previous breakpoint
breakpoint /after:5 at TOY\%LINE 15          !overwritten.
```

Note that the following commands overwrite one another when they contain identical address-expression parameters:

- SET BREAK (in all its forms)
- SET TRACE
- SET WATCH

5.2.1.1 The BREAK/AFTER:n Option - By specifying /AFTER:n as a command qualifier in the SET BREAK command, you can direct the debugger to count but not break at the first n-1 activations of the break location and to take break action only on the nth and subsequent activations of that location.

For example, as a result of the command:

```
DBG> SET BREAK/AFTER:10 LABL
```

a breakpoint occurs on the 10th activation of LABL. All subsequent activations of LABL result in breakpoints -- for example, the 11th activation, the 12th, and so on.

Note that the command SET BREAK/AFTER:1 is identical in effect to the command SET BREAK.

The following example demonstrates the use of the /AFTER:n command qualifier:

```
DBG> SET BREAK/AFTER:2 %LINE 15         !Break on 2nd pass at line 15.

DBG> SET BREAK %LINE 16                 !Break at line 16.

DBG> SHOW BREAK                         !Two breakpoints in effect.
breakpoint at TOY\%LINE 16
breakpoint /after;2 at TOY\%LINE 15

DBG> GO                                 !Begin program execution
routine start at TOY                    !at the beginning of routine
break at TOY\%LINE 16                   !TOY.  Break at line 16.
                                        !Note that no break occurred
                                        !on first activation of line
                                        !15.

DBG> GO %LINE 14                        !Resume execution at line 14.
start at TOY\%LINE 14
break at TOY\%LINE 15                   !Breakpoint at line 15
                                        !occurs on second pass.
```

5.2.1.2 DO Command Sequence at Breakpoint - If you want the debugger to execute one or more debugger commands immediately after a breakpoint has been reached, you specify these commands in a DO command sequence as shown in the following format:

```
SET BREAK address-expression DO (command [;command...] )
```

Whether the DO command sequence is a single command, a list of commands, or a command procedure, parentheses are required delimiters, as shown in the format above. If the DO command sequence is a list of commands, you must separate commands with a semicolon (;).

The debugger executes commands in a DO command sequence in the order in which they are listed. If one of the commands in a DO command sequence is a command procedure (see Section 6.2), the debugger begins execution of the commands in that file when it reaches the command file specification (@file-spec) in the DO command sequence. After the debugger has executed each command in the command procedure, it continues executing commands in the DO command sequence until it exhausts them or until it reaches another command file specification.

The DO command sequence may contain commands that resume program execution. For example, a typical DO command sequence might contain several EXAMINE commands followed by a GO command.

Note that the debugger does not check the syntactical correctness of the commands within a DO command sequence after you issue the SET BREAK command containing that DO command sequence; the debugger checks for syntactical correctness when it actually executes the DO command sequence at breakpoint activation.

Any debugger command, including a SET BREAK command with a DO command sequence, may appear in a DO command sequence. Nesting of DO command sequences is permissible to any level, so long as syntax rules have not been violated.

You are not limited in the number and type of debugger commmands you can include in a DO command sequence. If all the commands you want to include in the sequence do not fit on a single input line, specify the line continuation character (-) at the end of the input line and then press RETURN. In this way, you can continue entering commands within the same DO command sequence.

The following example demonstrates the power and versatility of the DO command sequence at breakpoint:

```
DBG> SET BREAK %LINE 16 DO (EXAMINE I;  EXAMINE R;  STEP)

DBG> GO
routine start at TOY                   !Start at routine.
break at TOY\%LINE 16                  !Break at line 16.
TOY\I:  1                              !EXAMINE I in DO sequence.
TOY\R:  0.0000000E+00                  !EXAMINE R in DO sequence.
start at TOY\%LINE 16
stepped to TOY\%LINE 17                !STEP in DO sequence.

DBG> SET BREAK %LINE 16 DO (EXAMINE I;EXAMINE R;  GO %LINE 15)

                                       !This command causes the
                                       !execution of an infinite
                                       !loop once the breakpoint at
                                       !line 16 is activated.

DBG> GO %LINE 15                       !Begin execution.
start at TOY\%LINE 15
break at TOY\%LINE 16                  !Breakpoint activated.
TOY\I:  1                              !EXAMINE I.
TOY\R:  0.0000000E+00                  !EXAMINE R.
start at TOY\%LINE 15                  !GO %LINE 15.
break at TOY\%LINE 16                  !Breakpoint activated.
TOY\I:  1                              !EXAMINE I.
TOY\R:  0.0000000E+00                  !EXAMINE R.
start at TOY\%LINE 15                  !GO %LINE 15.
break at TOY\%LINE 16                  !Breakpoint activated.
TOY\I:  1                              !and so on.
TOY\R:  0.0000000E+00
start at TOY\%LINE 15
break at TOY\%LINE 16

^Y                                     !Interrupt by CTRL/Y.

$
```

5.2.2 Exception Breakpoints

An event arising within the context of an executing program may require the execution of software outside that program's explicit flow of control. The notification of such an event is called an exception, and the presence of an exception indicates that a "condition" exists. Exception conditions range in severity and vary in the way they affect an executing program. The following are examples of exception conditions:

- An arithmetic overflow or underflow
- A memory access violation
- An invalid operation code
- A division by zero

You direct the debugger to treat an exception generated by your program as a breakpoint by issuing the command:

```
DBG> SET EXCEPTION BREAK
```

As a result of this command, whenever your program generates an exception condition, the debugger responds by suspending program execution, reporting the exception condition, and prompting you for input.

Whenever an exception breakpoint is activated, therefore, you have the opportunity to issue debugger commands. When you want to continue program execution, you can specify one of the following commands:

- A GO command without an address-expression parameter. In this case, the debugger fields and resignals the exception, thus allowing any user-declared exception handlers to execute.
- A GO command with an address-expression parameter. In this case, the debugger allows program execution to continue at the specified location, thus inhibiting the execution of any user-declared exception handlers.

Note that you cannot issue a STEP command to resume program execution after breakpoint activation. The STEP command is illegal in this context and, if issued, will result in a warning message.

If you do not specify an exception breakpoint by the SET EXCEPTION BREAK command or if you cancel an exception breakpoint by the CANCEL EXCEPTION BREAK command, exception conditions generated by your program are handled in the following way:

1. The debugger fields and resignals the exception.
2. If you have defined a condition handler in your program, it is executed.
3. If you have not defined a condition handler or if a condition handler that you have defined resignals the exception condition, a diagnostic message is issued and control is returned to the debugger, which then displays its prompt.

Note that an exception handler executes until one of the following occurs:

- The exception handler "handles" the condition, thus allowing the program to continue execution.
- The exception handler resignals the exception condition.
- The exception handler encounters a breakpoint or watchpoint.
- The exception handler generates its own exception.
- The exception handler exits, thus terminating program execution.

You cancel the SET EXCEPTION BREAK command by issuing the command:

```
DBG> CANCEL EXCEPTION BREAK
```

Note that the command CANCEL BREAK/ALL does not cancel an exception breakpoint.

For more information on exceptions, consult any of the following sources: VAX Architecture Handbook, VAX/VMS System Services Reference Manual, and VAX-11 Run-Time Library Reference Manual.

5.2.3 Watchpoints

You can direct the debugger to watch an entity and notify you if its value is modified. To do this, you specify as a parameter in the SET WATCH command an address expression that identifies the location where the entity resides. The format of the SET WATCH command is:

```
SET WATCH address-expression
```

As a result of the SET WATCH command, the debugger sets a watchpoint at the program location specified by the address expression.

If the entity has a compiler-generated type, the debugger uses the length in bytes associated with that type to determine the length in bytes of the watched location. If the entity does not have a compiler-generated type, the debugger watches four bytes of virtual memory, beginning at the byte identified by the address expression.

Whenever an instruction causes the modification of a watched entity, the debugger:

1. Suspends program exection after that instruction has completed execution
2. Identifies the watched entity
3. Identifies the instruction that modified the entity
4. Reports the old value of the entity
5. Reports the new (modified) value of the entity
6. Issues its prompt

To display watchpoints currently in effect, issue the command:

```
DBG> SHOW WATCH
```

To cancel a watchpoint, issue the CANCEL WATCH command in the following format:

```
CANCEL WATCH [/qualifier] [address-expression]
```

If you specify an address-expression parameter, the watchpoint at the location denoted by the address expression is canceled. In this case, you cannot also specify a command qualifier.

If you specify the /ALL command qualifier, all watchpoints are canceled. In this case, you cannot also specify an address-expression parameter.

The following example demonstrates the SET WATCH, SHOW WATCH, and CANCEL WATCH commands:

```
DBG> SET WATCH RESULT                  !Set watchpoint at RESULT.

DBG> SHOW WATCH                                  !Display watchpoints.
watchpoint at TOTAL\RESULT for 4.  bytes         !There are 4 bytes
                                                 !watched beginning at
                                                 !TOTAL\RESULT.

DBG> GO                                      !Notification of
start at TOTAL\START+02                      !watchpoint modification.
write to TOTAL\RESULT at PC TOTAL\GO3        !Note that TOTAL\GO3
old value = 0000000                          !identifies the location
new value = 00000A66                         !of the instruction that
                                             !caused the modification
                                             !of RESULT.

DBG> SHOW MODE                         !Values are displayed in
modes:  symbolic, hexadecimal          !hexadecimal. Address
                                       !expressions are displayed
                                       !symbolically where possible.

DBG> SHOW WATCH                                  !Watchpoint still in
watchpoint at TOTAL\RESULT for 4. bytes.         !effect.

DBG> CANCEL WATCH RESULT               !Cancel the watchpoint.

DBG> SHOW WATCH                        !Show current watchpoints.
%DEBUGI-NOWATCHES, no watchpoints are set
```

5.2.3.1 Watchpoint Restrictions - The entity identified by the address expression in a SET WATCH command must reside in the P0 region of virtual memory. The P0 region is that part of process memory space where program images and most of their data reside. Note that the stack is not in P0 space.

Typically the entity identified by the address expression in a SET WATCH command cannot be dynamically allocated; that is, the allocation of memory storage to that variable must take place at compile time, not at run time.

The mechanism for implementing watchpoints is to establish write protection for the page of memory that contains the watchpoint; consequently, when a write operation to that page is attempted, an exception occurs.

If the write operation is initiated by a system service or by VAX-11 RMS, failure status is returned, usually with severe implications for the program. If the write operation is not initiated by a system service or by VAX-11 RMS, then the debugger either initiates watchpoint action if the write is to a watched location or allows the re-execution of the instruction that caused the exception, thus permitting the write operation to take place without generating a second exception.

In sum, you should not set a watchpoint on an entity that resides on a page in memory that might be referenced by a system service or by VAX-11 RMS. In particular, since system services that perform input/output read from and write to pages of virtual memory, it is wise to cancel watchpoints before input/output operations are performed.

Finally, in some circumstances, watchpoints may not be signaled due to certain hardware restrictions and/or the effects of compiler optimization.

5.3 MONITORING PROGRAM EXECUTION

This section discusses how to monitor program execution by tracing the flow of program control and by displaying the current state of procedure calls.

5.3.1 Tracepoints

By setting tracepoints, you can monitor the sequence in which instructions in your program are executed and thereby check for unexpected control transfers. Moreover, tracepoints do not interrupt or otherwise disturb program execution.

When a tracepoint is activated, the debugger:

1. Suspends execution at the traced location
2. Reports that execution has reached the traced location
3. Resumes execution at the point of suspension

To set a tracepoint, issue the SET TRACE command in the following format:

```
SET TRACE [/qualifier] [address-expression]
```

If you specify an address-expression parameter, the debugger sets a tracepoint at the location denoted by the address expression. In this case, you cannot also specify a command qualifier.

If you specify the /CALL or the /BRANCH command qualifier, the debugger sets tracepoints at all instructions that are members of the family of CALL or BRANCH instructions, respectively (see Section 5.3.1.1). In this case, you cannot also specify an address-expression parameter.

To see what tracepoints are in effect, issue the command:

```
DBG> SHOW TRACE
```

To cancel one or more tracepoints, issue the CANCEL TRACE command in the following format:

```
CANCEL TRACE [/qualifier] [address-expression]
```

If you specify an address-expression parameter, the debugger cancels the tracepoint at the location denoted by the address expression. In this case, you cannot also specify a command qualifier.

If you specify the /CALL or the /BRANCH command qualifier, the debugger cancels tracepoints at all instructions that are members of the family of CALL or BRANCH instructions, respectively. In this case, you cannot also specify an address-expression parameter.

If you specify the /ALL command qualifier, the debugger cancels all tracepoints. In this case, you cannot also specify an address-expression parameter.

The following demonstrates the use of the SET, SHOW, and CANCEL TRACE commands:

```
DBG> SHOW TRACE                       !Show tracepoints in effect.
%DEBUG-I-NOTRACES, no tracepoints are set, no opcode tracing

DBG> SET TRACE %LINE 30               !Set tracepoint at %LINE 30.

DBG> SET BREAK %LINE 60               !Set breakpoint at %LINE 60.

DBG> SHOW TRACE                       !Show tracepoints in effect.
tracepoint at MEANSUB$MAIN\%LINE 30

DBG> GO                               !Begin program execution.
routine start at MEANSUB$MAIN         !Start of program.
trace at MEANSUB$MAIN\%LINE 30        !Tracepoint reached.
break at MEANSUB$MAIN\%LINE 60        !Breakpoint reached.

DBG> SET TRACE %LINE 40               !Set another tracepoint.

DBG> SHOW TRACE                       !Show tracepoints in effect.
tracepoint at MEANSUB$MAIN\%LINE 40
tracepoint at MEANSUB$MAIN\%LINE 30

DBG> GO %LINE 20                      !Resume execution at %LINE 20.
start at MEANSUB$MAIN\%LINE 30
trace at MEANSUB$MAIN\%LINE 30        !Tracepoint reached.
trace at MEANSUB$MAIN\%LINE 40        !Tracepoint reached.
break at MEANSUB$MAIN\%LINE 60        !Breakpoint reached.

DBG> CANCEL TRACE %LINE 40            !Cancel one tracepoint.

DBG> SHOW TRACE                       !Show tracepoints in effect.
tracepoint at MEANSUB$MAIN\%LINE 30

DBG> CANCEL TRACE/ALL                 !Cancel all tracepoints.

DBG> SHOW TRACE                       !Show tracepoints in effect.
%DEBUG-I-NOTRACES, no tracepoints are set, no opcode tracing
```

5.3.1.1 Opcode Tracing - Opcode tracing is the tracing of one or both of the two families of instructions: CALL and BRANCH.

If you issue the command:

```
DBG> SET TRACE/CALL
```

the debugger sets a tracepoint at each of the following instructions: CALLS, CALLG, BSBW, BSBB, JSB, RSB, RET.

If you issue the command:

```
DBG> SET TRACE/BRANCH
```

the debugger sets a tracepoint at each of the following instructions: BNEQ, BEQL, BGTR, BLEQ, BGEQ, BLSS, BGTRU, BLEQU, BVC, BVS, BGEQU, BLSSU, BRB, BRW, JMP, BBS, BBC, BBSS, BBCS, BBCC, BBSSI, BBCCI, BLBS, BLBC, ACBB, ACBW, ACBL, ACBQ, ACBD, ACBG, ACBH, AOBLEQ, AOBLSS, SOBGEQ, SOBGTR, CASEB, CASEW, CASEL.

Notice that you do not specify an address-expression parameter in the SET TRACE command when you use the /CALL and /BRANCH qualifiers. The use of either of the SET TRACE qualifiers is called opcode tracing because tracepoints are set on particular instructions (thus on particular instruction op codes).

To see the tracepoints currently in effect, issue the command:

```
DBG> SHOW TRACE
```

To cancel tracepoints at CALL or BRANCH instructions, use the CANCEL TRACE command with the appropriate qualifier, as follows:

```
DBG> CANCEL TRACE/CALL

DBG> CANCEL TRACE/BRANCH
```

Note that opcode tracing noticeably slows program execution.

The following example demonstrates the use of the /CALL and /BRANCH qualifiers in the SET, SHOW, and CANCEL TRACE commands:

```
DBG> SET TRACE/CALL                      !Trace call instructions.

DBG> SHOW TRACE                          !Show tracepoints in effect.
tracing /CALL instructions:  CALLS, CALLG, BSBW, BSBB, JSB, RSB
and RET

DBG> GO %LINE 20                         !Begin program execution.
start at MEANSUB$MAIN\%LINE 30
trace at PC MEANSUB$MAIN\%LINE 30 +9: CALLS \#1,L^BAS$INIT_GOSUB
trace at PC 24566:  JSB    @B^4(AP)
trace at PC MEANSUB$MAIN\%LINE 200 +7: CALLS \#1,L^BAS$PRINT
trace at PC 24918: CALLS  #4,W^24987
trace at PC 25003: JSB  L^8352
trace at PC 8407: JSB  L^94721
trace at PC 94877: RSB
trace at PC 8492: RSB
trace at PC 25190: JSB  L^34295
trace at PC 34333: RSB

          .
          .
          .

DBG> SHOW TRACE                          !Show tracepoints in effect.
tracing /CALL instructions:  CALLS, CALLG, BSBW, BSBB, JSB, RSB
and RET
```

```
DBG> SET TRACE/BRANCH                   !Trace all branch
                                        !instructions.

DBG> SHOW TRACE                         !Show tracepoints in effect.
tracing /CALL instructions:  CALLS, CALLG, BSBW, BSBB, JSB, RSB
and RET
tracing /BRANCH instructions:  BNEQ, BEQL, BGTR, BLEQ, BGEQ,
BLSS, BGTRU,
        BLEQU, BVC, BVS, BGEQU, BLSSU, BRB, BRW, JMP, BBS, BBC,
BBSS, BBCS,
        BBSC, BBCC, BBSSI, BBCCI, BLBS, BLBC, ACBB, ACBW, ACBL,
ACBF, ACBD,
        ACBG, ACBH, AOBLEQ, AOBLSS, SOBGEQ, SOBGTR, CASEB, CASEW
and CASEL

DBG> GO %LINE 20                        !Resume program execution at
start at MEANSUB$MAIN\%LINE 30          !%LINE 20.
trace at PC MEANSUB$MAIN\%LINE 30 +9: CALLS \#1,L^BAS$INIT_GOSUB
routine trace at PC BAS$INIT_GOSUB: JMP  L^24510
trace at PC 24552: BBC  #11,B^-26(R0),24559
trace at PC 24566: JSB  @B^4(AP)
trace at PC MEANSUB$MAIN\%LINE 200 +7: CALLS \#1,L^BAS$PRINT
routine trace at PC BAS$PRINT: JMP  L^24896
trace at PC 24906: BNEQ  24913
trace at PC 24911: BRB  24916
trace at PC 24918: CALLS  #4,W^24987
trace at PC 25003: JSB  l^8352
trace at PC 8359: BGTR  8366
trace at PC 8364: BGEQ  8377
trace at PC 8391: BEQL  8407
trace at PC 8407: JSB  L^94721
trace at PC 94728: BLBS  L^103336,94740
trace at PC 94752: BBCS  R2,L^104364,94762
trace at PC 94772: BEQL  94790
trace at PC 94804: BEQL  94817
trace at PC 94815: BRB  94829
trace at PC 94860: BLBC  B^4(SP),94869
trace at PC 94877: RSB
trace at PC 8413: CASEL  R0,#1,#2
                                        8431,
                                        8431,
                                        8458
trace at PC 8437: BNEQ  8444

           .
           .
           .

DBG> CANCEL TRACE/ALL                   !Cancel all tracepoints.

DBG> SHOW TRACE                         !Show tracepoints in effect.
%DEBUG-I-NOTRACES, no tracepoints are set, no opcode tracing
```

5.3.2 The SHOW CALLS Command

When procedure A calls procedure B (by means of a CALL instruction), VAX/VMS preserves information about the program state of procedure A at the time that control is transferred to procedure B. This information is stored on the stack in a call frame for procedure B. The execution of a CALL instruction, therefore, results in the construction of a call frame for the called routine; this call frame contains information about the calling routine.

If procedure B calls procedure C, VAX/VMS builds a call frame for procedure C. The call frame for procedure C is built on top of the call frame for procedure B. In other words, the call frame for the most recent called procedure is on the top of the stack.

When a routine returns to its caller, the call frame for that routine is removed from the stack. Thus when procedure C finishes and control is returned to procedure B, the call frame for procedure C is removed from the stack.

The SHOW CALLS command provides information about the sequence of currently active procedure calls or the number of call frames on the stack. For example, if your program contains a recursive routine, you can use a SHOW CALLS command to examine the chain of recursion.

The format of the SHOW CALLS command is:

```
SHOW CALLS [n]
```

The optional parameter n specifies the call count, or the number of call frames to be displayed, by a decimal integer in the range 0 through 32767. If you do not specify the parameter n, then information on all call frames is displayed. If the call count represented by n exceeds the current number of call frames, information on all call frames is displayed. If the call count is 0, the command is accepted but no information is displayed. Otherwise, the number of call frames specified by the parameter n is displayed.

For each call frame, the debugger displays one line of information. The first line displayed contains information about the top call frame (the one representing the most recently called procedure); the next line contains information on the next most recently called procedure; and so on.

Each line of information displayed by the debugger contains:

- The name of the module that contains the called routine.
- The name of the called routine.
- The line number of the call (in line-oriented languages only).
- The value of the PC in the calling routine at the time that control was transferred to the called routine. Note that this value is the location of the instruction following the call. The value of the PC is expressed in two ways: as an absolute virtual address and as a virtual address relative to the virtual address of the name of the routine.

Note that even if your program contains no procedure calls, the debugger displays an active call when you issue the SHOW CALLS command. The reason for this is that your program has a stack frame built for it when it is first activated.

Thus if the debugger responds that there are no active calls when you issue a SHOW CALLS command, either your program has terminated or the stack has been corrupted.

The following example demonstrates the use of the SHOW CALLS command:

```
DBG> STEP
start at SUB2\%LINE 15
stepped to SUB2\%LINE 16
```

```
DBG> SHOW CALLS                          !Display active procedure calls.

module name     routine name        line    rel PC      abs PC

SUB2            SUB2                        00000002    0000085A
SUB1            SUB1                  5     00000014    00000854
MAIN            MAIN                 10     0000002C    0000082C

                                     !SUB2 is the procedure that is
                                     !currently executing.  SUB2
                                     !was called by SUB1, and SUB1
                                     !was called by MAIN.

DBG> GO                              !Continue program execution.
start at 19351
%DEBUG-I-EXITSTATUS, is '%SYSTEM-S-NORMAL, normal successful
completion'

DBG> SHOW CALLS
%DEBUG-W-NOCALLS, no active call frames
                                     !Program has terminated.
```

5.4 EXIT HANDLERS

Exit handlers are procedures that are called whenever an image requests the $EXIT system service. A user program may declare one or more exit handlers. The debugger always declares its own exit handler.

This section discusses the sequence in which exit handlers are executed and explains how to debug them.

5.4.1 Sequence of Exit Handler Execution

At program termination, exit handlers are executed in last-in/first-out order; that is, the most recently declared exit handler is executed first, then the next most recently declared exit handler, and so on. However, if an exit handler terminates by the $EXIT system service rather than by a RET, previously declared exit handlers (those that are first-in relative to it) are not executed.

Since the debugger exit handler terminates by $EXIT, user exit handlers that are declared before the debugger is invoked will not be executed because they are first-in relative to it. This can occur if the debugger is invoked by means of the DEBUG command following program interruption.

When the debugger is activated by a RUN command in the typical fashion, that is, by LINK/DEBUG followed by RUN, or simply by RUN/DEBUG, the debugger exit handler is declared before control passes to the user program; that is, the debugger exit handler is the first-in and therefore the last-out. Consequently, at program termination, the debugger exit handler executes after all user-declared exit handlers have executed.

However, when the debugger is activated by the DEBUG command following program interruption by CTRL/Y or CTRL/C, the debugger exit handler may not be the first-in. Exit handlers declared by the user program before program interruption will precede the debugger exit handler in

the queue; that is, they will be first-in and therefore last-out relative to the debugger exit handler. Consequently, at program termination, the debugger exit handler will execute before these user-declared exit handlers, thereby inhibiting their execution (because the debugger terminates by $EXIT). On the other hand, exit handlers declared by the user program after program interruption will be executed because they are last-in and therefore first-out relative to the debugger exit handler.

5.4.2 Debugging Exit Handlers

To debug an exit handler, you must first set a breakpoint in that exit handler. Then, you must cause that exit handler to execute either by including in your program an instruction that invokes the exit handler or by allowing your program to terminate. (At program termination, the system begins executing exit handlers in first-in/last-out order.) When the exit handler executes, the breakpoint will be activated and control returned to the debugger, which will display its prompt. You can then enter debugger commands.

You cannot debug an exit handler if you terminate the debugging session with EXIT or CTRL/Z. When you enter EXIT or CTRL/Z, the debugger deactivates all breakpoints, tracepoints, and watchpoints in your program. The system then executes any declared exit handlers as described above; however, since breakpoints have been deactivated, control does not pass to the debugger.

CHAPTER 6

LOG FILES AND COMMAND PROCEDURES

You can direct the debugger to keep a record of a debugging session by creating a log file. You can also direct the debugger to execute debugger commands listed in an external file. Such a file, when executed, is called a command procedure. Further, you can direct the debugger to execute a log file as a command procedure. This chapter discusses these topics.

6.1 LOG FILES

A debugger log file is a file containing every debugger command you issue during a particular debugging session, together with a display of the debugger's response to those commands.

In a debugger log file, any command you issue in response to the debugger prompt (DBG>) begins a line, but the debugger prompt itself is not recorded. The debugger's response to the command appears on the following line or lines and, except for an exclamation point at the beginning of each line of response, is identical to what you see at your terminal.

The following is an example of a debugger log file. Note the absence of debugger prompts and the presence of exclamation points at the beginning of each line of debugger response.

```
SHOW OUTPUT
!output: noverify, terminal, logging to _DB2:[GMF]EV.LOG;1
SHOW TRACE
!%DEBUG-I-NOTRACES, no tracepoints are set, no opcode tracing
SET TRACE %LINE 30
SET BREAK %LINE 60
SHOW TRACE
!tracepoint at MEANSUB$MAIN %LINE 30
GO
!routine start at MEANSUB$MAIN
!trace at MEANSUB$MAIN\%LINE 30
!break at MEANSUB$MAIN\%LINE 60
SET TRACE %LINE 40
SHOW TRACE
!tracepoint at MEANSUB$MAIN\%LINE 40
!tracepoint at MEANSUB$MAIN\%LINE 30
GO %LINE 20
!start at MEANSUB$MAIN\%LINE 30
!trace at MEANSUB$MAIN\%LINE 30
!trace at MEANSUB$MAIN\%LINE 40
!break at MEANSUB$MAIN\%LINE 60
```

To create a debugger log file, use the SET OUTPUT command. To name a debugger log file, use the SET LOG command. The following subsections discuss these commands.

6.1.1 The SET OUTPUT and SHOW OUTPUT Commands

The SET OUTPUT command controls the debugger's current output configuration; that is, it controls the way in which the debugger's responses to commands you issue are displayed and recorded.

The format of the SET OUTPUT command is:

```
SET OUTPUT parameter [,parameter...]
```

The following is a list of output parameters that you may specify in the SET OUTPUT command:

- LOG -- Debugger output is written to a log file. If you have not used the SET LOG command to name a log file, then the debugger uses the default file name DEBUG.LOG.

- NOLOG -- Debugger output is not written to a log file. This is a default output parameter; the debugger does not write output to a log file unless you explicitly request it.

- TERMINAL -- Debugger output is displayed at the terminal. This is a default output parameter; the debugger displays output on the terminal unless you explicitly request that it not do so.

- NOTERMINAL -- Debugger output, except for diagnostic messages, is not displayed at the terminal.

- VERIFY -- The debugger includes, in its output, each input command string in any command procedure or DO command sequence that it is executing. That is, the debugger not only displays the results of executing each command in a command procedure or DO command sequence but also displays the commands that caused execution.

  The debugger displays commands from command procedures and DO command sequences in accordance with the current status of the output parameters LOG and TERMINAL. For example, if TERMINAL is set but LOG is not set, then the VERIFY parameter causes the display of commands from command procedures and DO command sequences to be displayed at the terminal.

- NOVERIFY -- The debugger does not include as output each input command string that it executes from a command procedure or from a DO command sequence.

  This is a default output parameter; the debugger does not display commands that it executes from command procedures or DO command sequences unless you explicitly direct it to do so.

To see which output parameters are currently in effect, issue the command:

```
DBG> SHOW OUTPUT
```

To change a current output parameter, issue a SET OUTPUT command specifying the desired output configuration.

When you issue the command SET OUTPUT LOG, the debugger creates a log file if one does not already exist. By default, the debugger creates a log file with the file specification DEBUG.LOG. The debugger assigns a version number one higher than that of any existing DEBUG.LOG files.

If you have previously used the SET LOG command to name a log file, the debugger opens a log file with that name when you issue the command SET OUTPUT LOG. See Section 6.1.2 for more information on the SET LOG command.

6.1.2 The SET LOG and SHOW LOG Commands

If the output parameter LOG is in effect (by SET OUTPUT LOG), you can direct the debugger to write output to a specified file by issuing the SET LOG command in the following format:

```
SET LOG file-spec
```

The file-spec parameter is the file specification of the file to which you want the debugger to write its output.

If the output parameter LOG is in effect but you have not issued the SET LOG command to name a log file, the debugger writes output to a file with the default name DEBUG.LOG.

If the output parameter LOG is not in effect, the debugger does not write its output to the file specified in the SET LOG command. However, if you subsequently issue a SET OUTPUT LOG command, the debugger begins writing output to the file specified in the SET LOG command.

If the file-spec parameter in the SET LOG command includes both a file name and a file type, the debugger writes to the file so specified.

If the file-spec parameter in the SET LOG command does not include a file type, the debugger uses the default file type of LOG.

To find out the name of the current log file, issue the command:

```
DBG> SHOW LOG
```

The following example demonstrates the SET LOG, SHOW LOG, SET OUTPUT, and SHOW OUTPUT commands:

```
DBG> SET LOG FILE                     !Name a log file.

DBG> SET OUTPUT LOG                   !Write output to the log file.

DBG> SHOW OUTPUT
output: noverify, terminal, logging to _DB2:[GMF]FILE.LOG;1
                                      !Display current output
                                      !configuration.

DBG> SHOW LOG                         !Display name of current log
logging to _DB2:[GMF]FILE.LOG;1       !file.

DBG> SET OUTPUT NOTERMINAL,NOLOG
%DEBUG-I-OUTPUTLOST, output being lost, both NOTERMINAL and NOLOG
are in effect

DBG> SHOW LOG
not logging to _DB2:[GMF]FILE.LOG;1

DBG> SET LOG TERT.EEE                 !Name a new log file.
```

```
DBG> SHOW LOG                                !Not logging to TERT.EEE
not logging to _DB2:[GMF]TERT.EEE;1          !because output is set
                                             !to NOLOG.

DBG> SET OUTPUT LOG                        !Change output configuration.

DBG> SHOW LOG
logging to _DB2:[GMF]TERT.EEE;1

DBG> SHOW OUTPUT
output: noverify, noterminal, logging to _DB2:[GMF]TERT.EEE;1
                                           !Display current output
                                           !configuration.
```

6.2 COMMAND PROCEDURES

A command procedure is a file containing one or more debugger commands. You can cause the debugger to execute debugger commands from a file by prefixing the file specification of that file with the "at sign" (@), in the following format:

```
@file-spec
```

If the file specification includes a file name but not a file type, the debugger assumes the default file type COM.

The @file-spec command may be issued from the terminal, from within a DO command sequence, or from within another command procedure.

If the @file-spec command is issued from the terminal, the debugger begins with the execution of the first command in the file and displays its prompt after all commands in the file have been executed.

If the @file-spec command is one of the commands in a DO command sequence, the debugger begins execution of the first command in the file upon reaching the @file-spec command in the DO command sequence. After execution of all commands in the file, the debugger resumes execution of any remaining commands in the DO command sequence and then issues its prompt.

If the @file-spec command is one of the commands in another (calling) command procedure, the debugger begins execution of the commands in the called command procedure when it reaches the @file-spec command. It resumes execution of any remaining commands in the calling command procedure after all commands in the called command procedure have been executed.

If a command in a command procedure is syntactically incorrect, the debugger issues an error message and continues execution with the next command in the command procedure.

When the default output parameter NOVERIFY is in effect (see Section 6.1.1), the debugger does not display -- either on the terminal or in a log file -- the commands in a command procedure. The debugger simply executes the commands and displays the resulting output.

If you want the debugger to display the commands in a command procedure as it executes them, you must set the output parameter to VERIFY (see Section 6.1.1).

6.2.1 Editor-Created Command Procedures

You can use an editor to create a command procedure. You do this by creating a file and then entering debugger commands into the file.

For example, the following debugger commands make up the file whose file specification is BREAK.EDT:

```
SET BREAK/AFTER:3 %LINE 120 DO (EXAMINE K,N,J,X(K); GO)
SET BREAK/AFTER:3 %LINE 160 DO (EXAMINE K,N,J,X(K),S; GO)
SET BREAK %LINE 90
```

The purpose of the commands in this command procedure is to set breakpoints at three different program locations. Thus, whenever the command:

```
DBG> @BREAK.EDT
```

is issued in the debugging session, the debugger sets the breakpoints specified in the command procedure.

The following example is a log file BREAK.LOG that records the use of the command procedure BREAK.EDT in a debugging session:

```
@BREAK.EDT                    !Cause command procedure to be
                              !executed.
SHOW BREAK                    !Show breakpoints set by @BREAK.EDT.
!breakpoint at MEANSUB$MAIN\%LINE 90
!breakpoint /after:3 at MEANSUB$MAIN\%LINE 160 DO (EX
K,N,J,X(K),S; GO)
!breakpoint /after:3 at MEANSUB$MAIN\%LINE 120 DO (EX K,N,J,X(K);
GO)
GO                            !Begin program execution.
!routine start at MEANSUB$MAIN\MEANSUB$MAIN
!break at MEANSUB$MAIN\MEANSUB$MAIN %LINE 120
!MEANSUB$MAIN\MEANSUB$MAIN\K: 3.000000
!MEANSUB$MAIN\MEANSUB$MAIN\N: 3.000000
!MEANSUB$MAIN\MEANSUB$MAIN\J: 1.000000
!MEANSUB$MAIN\MEANSUB$MAIN\X(3):  93.00000
!start at MEANSUB$MAIN\MEANSUB$MAIN\%LINE 120
!break at MEANSUB$MAIN\MEANSUB$MAIN\%LINE 160
!MEANSUB$MAIN\MEANSUB$MAIN\K: 3.000000
!MEANSUB$MAIN\MEANSUB$MAIN\N: 3.000000
!MEANSUB$MAIN\MEANSUB$MAIN\J: 1.000000
!MEANSUB$MAIN\MEANSUB$MAIN\X(3):  93.00000
!MEANSUB$MAIN\MEANSUB$MAIN\S: 252.0000
!start at MEANSUB$MAIN\MEANSUB$MAIN\%LINE 160
!break at MEANSUB$MAIN\MEANSUB$MAIN\%LINE 90
CANCEL BREAK/ALL            !Cancel all breakpoints.
```

6.2.2 Using Log Files as Command Procedures

You can use a debugger log file as a command procedure. A debugger log file contains debugger commands and debugger output; however, the debugger output is always preceded on the line by an exclamation point, which signals the debugger to ignore the line.

The exclamation point (!) is called the comment delimiter because it is used to delimit comments within the command string. The debugger's use of the comment delimiter on its own output in log files makes possible the direct use, without modification, of debugger log files as command procedures.

You request that the debugger execute commands from a log file by specifying that file as a command procedure, as follows:

```
DBG> @file-spec
```

Thus, the log file BREAK.LOG shown in Section 6.2.1 may be used as a command procedure in another debugging session by issuing the command:

```
DBG> @BREAK.LOG
```

The issuing of the command @BREAK.LOG in a subsequent debugging session causes the following debugger commands to be executed:

- @BREAK.EDT
- SHOW BREAK
- GO
- CANCEL BREAK/ALL

Thus, you can reexecute a previous debugging session by using a log file as a command procedure. This is helpful if you want to continue debugging from where you left off at a previous session. You can also use the log file as a command procedure in another program or in a modified version of the same program.

CHAPTER 7

ASSEMBLY-LEVEL DEBUGGING

Assembly-level debugging may involve one or more of the following:

- Treating program data as if it had no associated compiler-generated type
- Interpreting program data in octal or hexadecimal radix rather than in decimal radix because these radixes more accurately portray bit configurations in virtual memory addresses
- Referring to data using virtual addresses rather than symbols
- Referring to program locations as virtual addresses or as offsets to program labels
- Examining and manipulating the contents of machine-level objects such as registers
- Controlling and monitoring program execution by instruction rather than by line
- Examining, depositing, and replacing instructions

In the event that you need to debug a high-level language program (or a VAX-11 MACRO program) at the assembly level, you can use the following debugger features:

- A variety of types in which you can display the contents of virtual memory addresses
- Octal and hexadecimal radix modes in which you can interpret the contents of virtual memory addresses
- Nonsymbolic mode by which you can translate any symbol into its numeric equivalent
- Debugger permanent symbols by which you can refer to machine registers
- The STEP/INSTRUCTION command, which enables you to see every assembly-level instruction in your program as you step through it
- The SET TRACE/CALL and SET TRACE/BRANCH commands, which enable you to trace program flow by monitoring all instructions that alter the normal "inline" flow of control

Though most of these features are discussed elsewhere in the manual, they are discussed again in this chapter to facilitate quick reference by a programmer who wants to debug at the assembly level.

7.1 TYPE

You can display program data in any of the following debugger types: BYTE, WORD, LONGWORD, QUADWORD, OCTAWORD, FLOAT, D_FLOAT, G_FLOAT, H_FLOAT, ASCII:n, and INSTRUCTION.

If your program does not contain data with associated compiler-generated types, you can specify that all data be displayed in one of the debugger types by specifying the desired type in the SET TYPE command.

For example, the command:

```
DBG> SET TYPE ASCII:6
```

requests that any untyped reference be interpreted as a 6-byte ASCII string. This command changes the default type to ASCII:6.

If you intend to debug your program using one of the debugger types but your program contains data with associated compiler-generated types, you can inhibit the use of these compiler-generated types by specifying the desired debugger type as an override type in the SET TYPE/OVERRIDE command.

As a result of using this command, all data is displayed in the specified override type unless you use another override type as a command qualifier or cancel the override type using the CANCEL TYPE/OVERRIDE command.

For example, the command:

```
DBG> SET TYPE/OVERRIDE INSTRUCTION
```

directs the debugger to display the contents of a program location as a VAX-11 instruction.

If, in a particular command, you want to enter or display data in a type other than the current default or override type, you can specify that type as a command qualifier. A type specified as a command qualifier is a command override type. When you use a command override type, data referenced by that command is entered or displayed in that type for the duration of the command. After command execution, the current default or override type is in effect.

For example, the command:

```
DBG> EXAMINE/BYTE 1024
```

directs the debugger to display the byte at virtual memory address 1024 as an integer in the current radix mode, even if some other type (for example, longword integer) is currently in effect.

In sum, you can control the type used by the debugger to interpret the value of an entity in three ways, listed below in order of decreasing power:

1. Type command qualifier
2. SET TYPE/OVERRIDE command
3. SET TYPE command

To see what type is in effect, issue the command:

```
DBG> SHOW TYPE
```

To see what override type is in effect, issue the command:

```
DBG> SHOW TYPE/OVERRIDE
```

See Chapter 4 for more examples of these types and how they are used in debugger commands.

7.2 MODES

Modes influence the debugger's interpretation of information you enter in debugger commands, as well as the debugger's display of the results of command execution.

You establish a mode by using the SET MODE command or by specifying a mode as a command qualifier. If you do not explicitly specify a mode, the debugger assumes certain default modes, which depend on the current language setting.

There are two classes of mode:

- Radix mode, which influences the interpretation of numeric literals and, under certain circumstances, the display of data
- Symbolic or nonsymbolic mode, which influences the debugger's display of information under certain conditions

The following subsections discuss these modes.

7.2.1 Radix Modes

You can direct the debugger to interpret numeric literals and, under certain circumstances, display data in one of the three radix modes by issuing the command:

```
DBG> SET MODE radix-value
```

The parameter radix-value may be DECIMAL, HEXADECIMAL, or OCTAL.

For certain commands, you can override the current radix mode for the duration of a command by using a radix mode command qualifier. For example, if you have set the default radix mode to hexadecimal but want to have the value of a program location LOC displayed in octal, you specify that command qualifier as follows:

```
DBG> EXAMINE/OCTAL LOC
```

When the debugger evaluates a source-language expression or an address expression (in the EVALUATE or EVALUATE/ADDRESS commands, respectively), it interprets numeric literals within that expression in the current radix mode unless a radix mode command qualifier is specified. In that case it interprets numeric literals in the mode specified by the command qualifier.

When the debugger evaluates an address expression, it displays the result in the current radix mode unless a radix mode command qualifier is specified. In that case it displays the result in the radix mode specified by the command qualifier.

In some languages, when the debugger evaluates a source-language expression, it displays the result in the current radix mode unless a radix mode command qualifier is specified. In that case it displays the result in the mode specified by the command qualifier. In contrast, other languages may ignore radix qualifiers and display the result of an EVALUATE command in a radix determined by the source language.

The following example demonstrates how radix modes are used in the EXAMINE, EVALUATE, and EVALUATE/ADDRESS commands:

```
DBG>SHOW MODE                         !Display current modes.
modes: symbolic, decimal

DBG>SET MODE NOSYMBOL                 !Set mode to nonsymbolic.

DBG>SHOW TYPE                         !Display current type.
type: long integer

DBG>SET TYPE INSTRUCTION              !Set type to instruction.

DBG>EXAMINE  1035                     !Virtual address 1035 has a
1035:  MOVL    #1,B^44(R11)           !symbolic representation
                                      !(%LINE 15).  When mode is
                                      !nonsymbolic, %LINE 15 is
                                      !displayed as 1035.  Operands
                                      !are displayed in decimal.

DBG>EXAMINE/OCTAL .                      !Display nonsymbolic
00000002013:  MOVL    #001,B^054(R11) !representation of %LINE 15
                                         !in octal radix. Display
                                         !instruction operands in
                                         !octal radix.

DBG>EXAMINE/HEX .                     !Hexadecimal display of %LINE
0000040B:  MOVL    #01,B^2C(R11)      !15 and of instruction
                                      !operands.

DBG>EXAMINE  PSL                      !Display the PSL as a long-
PSL:  62914592                        !word in decimal radix.

DBG>EXAMINE/OCTAL  PSL                !Display the PSL in octal
PSL:  00360000040                     !radix.

DBG>SET MODE OCTAL                    !Set default radix mode to
                                      !octal.

DBG>SHOW MODE                         !Display current modes.
modes: nosymbolic, octal

DBG>EVALUATE/ADDRESS %LINE 15         !Display address of %LINE 15
00000002013                           !in octal (default) radix.

DBG>EV/A/DEC .                        !Display address of %LINE 15
1035                                  !in decimal radix.

DBG>EV/A/HEX .                        !Display address of %LINE 15
0000040B                              !in hexadecimal radix.

DBG>EV/A %LINE 15 - 10                !Perform address arithmetic
00000002003                           !in default octal radix.
```

```
DBG> EV 10 + 8
%DEBUG-W-INVNUMBER, invalid numeric string '8'
                                        !In octal radix, the character
                                        !"8" is not valid.

DBG> EV 7 + 1                           !Evaluate an expression in
00000000010                             !octal radix.

DBG> EV/HEX 7 + 1                       !Evaluate an expression in
00000008                                !hexadecimal radix.

DBG> SET MODE HEX                       !Set the default radix
                                        !to hexadecimal.

DBG> EV 9 + 2                           !Evaluate an expression in
0000000B                                !hexadecimal radix.

DBG> EV/A %LINE 15                      !Display address of %LINE 15
0000040B                                !in hexadecimal radix.

DBG> EV/A %LINE 15 - B
%DEBUG-W-NOSYMBOL, symbol 'B' is not in the symbol table

DBG> EV/A %LINE 15 - 0B !Prefix the "B" by a "0"
00000400                                !(zero) and the address
                                        !arithmetic is performed.
```

7.2.1.1 Radix Operators - By using a radix operator, you can direct the debugger to interpret a numeric literal in decimal (D), octal (O), or hexadecimal (X) radix, provided that the numeric literal is legal in the source language.

Radix operators are useful when you must enter several numeric literals of different radixes in a single debugger command. In this situation, a radix command qualifier alone is not adequate because it causes all numeric literals in the command to be interpreted in the specified radix. However, by using radix operators you can specify the radix in which each individual numeric literal is to be interpreted.

For example, assume you are depositing a VAX-11 instruction into a memory location denoted by an address expression that is a hexadecimal literal. Further, assume that one operand of the instruction is a decimal literal. You want the debugger to interpret these two numeric literals in their corresponding radixes. One way to do this is to use the /HEXADECIMAL radix command qualifier (to affect the address expression) and a decimal (D) radix operator (to affect the instruction operand). The following example demonstrates such a situation in VAX-11 MACRO:

```
DBG> DEPOSIT/HEXADECIMAL/INSTRUCTION 5432 = 'MOVL ^D222, R1
```

In this command, the debugger interprets the literal 5432 in hexadecimal radix and the literal 222 in decimal radix.

Note that radix operators may not be supported in all languages and that the syntax for specifying radix operators may vary from language to language. See the appropriate language documentation for more information.

7.2.2 Symbolic and Nonsymbolic Modes

If symbolic mode is set, the debugger displays information in its symbolic representation (if there is such a representation). If nonsymbolic mode is set, the debugger displays symbols in their equivalent numeric values in the current radix mode. Note that symbolic mode only affects the debugger's display; you can enter both symbolic or nonsymbolic values regardless of mode settings.

The symbolic and nonsymbolic modes influence the debugger's display of information in the three following circumstances:

1. **In the display of an address expression.** In symbolic mode, the debugger displays the value of an address expression in its symbolic representation, if any. In nonsymbolic mode, the debugger displays the value of an address expression as a virtual address, whether or not that virtual address is represented by a symbol. The following example demonstrates the effect of symbolic and nonsymbolic mode in the display of address expressions:

```
DBG> SHOW MODE                          !Display current modes.
modes: symbolic, decimal

DBG> EXAMINE J                          !Display address expression
TOY\J:  44444                           !J in full symbolic form;
                                        !that is, as TOY\J.

DBG> EXAMINE/NOSYMBOL J                 !Specify nonsymbolic mode.
712:  44444                             !J is displayed nonsymbol-
                                        !ically as address 712.

                                        !Mode does not affect the
                                        !interpretation of values you
                                        !enter. Here, J is an
                                        !acceptable operand in the
                                        !command, even though non-
                                        !symbolic mode is specified.
```

2. **In the display of the operands in an instruction.** In symbolic mode, any operand in an instruction is displayed in its symbolic representation (if it has one). In nonsymbolic mode, any operand in an instruction is displayed numerically, whether or not it has a symbolic representation. The following example demonstrates the effect of symbolic and nonsymbolic mode in the display of instruction operands:

```
DBG> SHOW MODE                          !Display current modes.
modes: symbolic, decimal

DBG>                                    !Display OUT as an instruction
DRAW\OUT:  CMPB    B^DRAW\COL,#0        !using the default symbolic
                                        !mode. Note that the
                                        !instruction operand DRAW\COL
                                        !is expressed symbolically.

DBG> EXAMINE/INSTRUCTION/NOSYMBOL OUT   !Specify NOSYMBOL mode. Note
3678:  CMPB     B^3599,#0               !that the instruction
                                        !operand DRAW\COL is now
                                        !expressed numerically.
```

```
DBG> EXAMINE/INSTRUCTION 3678          !In symbolic mode, you may
DRAW\OUT:  CMPB    B^DRAW\COL,#0       !enter information either sym-
                                       !bolically or nonsymbolically.
                                       !Address 3678 is the nonsym-
                                       !bolic location of OUT. Since
                                       !default mode is symbolic,
                                       !display is symbolic.

DBG> EXAMINE/INSTRUCTION/NOSYMBOL 3678 !When /NOSYMBOL is
3678:  CMPB    B^3599,#0               !specified, display is
                                       !nonsymbolic.
```

3. **In the display of the processor status longword (PSL).** In symbolic mode, execution of the command:

   ```
   DBG> EXAMINE PSL
   ```

   results in a formatted display of the contents of the PSL. In nonsymbolic mode, the PSL is displayed in numeric form. The following example demonstrates the effect of symbolic and nonsymbolic modes in the display of the PSL:

```
DBG> SHOW MODE                         !Display current modes.
modes: symbolic, decimal

DBG> EXAMINE PSL
PSL:     CMP TP FPD IS CURMOD PRVMOD IPL DV FU IV T N Z V C
         0   0  0   0  USER   USER   00  0  0  1  0 0 0 0 0

                                       !Symbolic mode gives formatted
                                       !display of PSL.

DBG> EXAMINE/NOSYMBOL PSL              !Display of PSL is not
PSL:  62914592                         !formatted.
```

7.3 PROGRAM CONTROL

At the assembly level, program locations are often specified by virtual addresses or by offsets from program labels, rather than by symbols. Thus, it is common to use virtual addresses as parameters in commands that affect program execution.

You can control and monitor program execution by means of all the commands discussed in Chapter 5: STEP, GO, CALL, SHOW CALLS, SET BREAK, SET TRACE, and SET WATCH. Refer to Chapter 5 for detailed descriptions of each of these commands.

This chapter discusses the use of two of these commands because they are very powerful tools in assembly-level debugging when used with certain command qualifiers. These commands are the STEP and the SET TRACE command.

7.3.1 Stepping Through the Program

You may find it useful to set the STEP parameter to INSTRUCTION. Then whenever you issue a STEP command, you will advance by one instruction or by the number of instructions you specify as a parameter in the STEP command. By single stepping, you can literally read your program's source code at the assembly level as VAX-11 instructions.

Further, you may want to set STEP parameters to INTO and to SYSTEM to enable you to step through the called routines in your program. Then whenever program execution reaches a routine call, you can follow program flow through the called routine (a single instruction at a time or by lines).

The following example demonstrates the use of these step parameters:

```
DBG> SHOW STEP              !Display current step parameters.
step type: nosystem, by line, over routine calls, nosource

DBG> SHOW MODE              !Display current modes.
modes: symbolic, decimal

DBG> SET STEP INSTRUCTION, INTO    !Set STEP parameters.

DBG> SHOW STEP              !Display current step parameters.
step type: nosystem, by instruction, into routine calls, nosource

DBG> STEP                   !Begin stepping through the program.
routine start at CIRCLE
stepped to CIRCLE %LINE 2 : PUSHAL  L^512

DBG> STEP
start at CIRCLE %LINE 2
stepped to CIRCLE %LINE 2 +6: MNEGL   #2,-(SP)

DBG> STEP
start at CIRCLE %LINE 2 +6
stepped to CIRCLE %LINE 2 +9: CALLS   #2,L^FOR$WRITE_SF

DBG> STEP
start at CIRCLE %LINE 2 +9
stepped to FOR$WRITE_SF+2: JMP    L^56758

                            !Step into a called routine
                            !named FOR$WRITE_SF.

                            !Once the 2-byte entry mask is
                            !stepped by, the debugger
                            !references instructions by
                            !their virtual address; this
                            !is because FOR$WRITE_SF is
                            !not in the Run-Time Symbol
                            !Table.

DBG> STEP
start at FOR$WRITE_SF+2
stepped to 56758: MOVZBL  #1,R0

DBG> STEP
start at 56758
stepped to 56761: BRW     56870

                 .
                 .
                 .

DBG> STEP 30
start at 55223
stepped to 58561: RET    !Return from FOR$WRITE_SF.
```

```
DBG> STEP
start at 58561
stepped to CIRCLE %LINE 4 : PUSHAL  L^535
                                    !Begin executing main program.

DBG> STEP
start at CIRCLE %LINE 4
stepped to CIRCLE %LINE 4 +6: MNEGL   #3,-(SP)

DBG> STEP
start at CIRCLE %LINE 4 +6
stepped to CIRCLE %LINE 4 +9: CALLS   #2,L^FOR$READ_SF

DBG> STEP
start at CIRCLE %LINE 4 +9
stepped to FOR$READ_SF+2: JMP     L^56646

                                    !Step into the called
                                    !routine FOR$READ_SF.
DBG> STEP
start at FOR$READ_SF+2
stepped to 56646: MOVZBL  #2,R0

DBG> STEP
start at 56646
stepped to 56649: BRW     56870

DBG> GO
start at 56649
%DEBUG-I-EXITSTATUS, is '%SYSTEM-S-NORMAL, normal successful
completion'
```

Note that the called routines in this example, FOR$WRITE_SF and FOR$READ_SF, are found in the user address space, not in system space. Stepping into a called routine in system space takes place in the same way as stepping into a called routine in the user space. The only difference is that the virtual memory addresses in system space have values greater than 80000000 hexadecimal.

7.3.2 Opcode Tracing

Opcode tracing is the tracing of either of two families of instruction: the family of CALL instructions and the family of BRANCH instructions. Since these instructions modify the "inline" flow of program execution, it is sometimes helpful to monitor their occurrence in a program.

You issue the command SET TRACE/CALL when you want to establish a tracepoint at all of the following instructions: CALLS, CALLG, BSBW, BSBB, JSB, RSB, RET.

As a result of the SET TRACE/CALL command, the debugger issues a message whenever it encounters one of the CALL instructions.

You issue the command SET TRACE/BRANCH when you want to establish a tracepoint at all of the following instructions: BNEQ, BEQL, BGTR, BLEQ, BGEQ, BLSS, BGTRU, BLEQU, BVC, BVS, BGEQU, BLSSU, BRB, BRW, JMP, BBS, BBC, BBSS, BBCS, BBCC, BBSSI, BBCCI, BLBS, BLBC, ACBB, ACBW, ACBL, ACBQ, ACBD, ACBG, ACBH, AOBLEQ, AOBLSS, SOBGEQ, SOBGTR, CASEB, CASEW, CASEL.

As a result of the SET TRACE/BRANCH command, the debugger issues a message whenever it encounters one of the BRANCH instructions.

Notice that you do not specify an address-expression parameter in the SET TRACE command when you use the /CALL and /BRANCH qualifiers. The use of either of the SET TRACE qualifiers is called opcode tracing because tracepoints are set on particular instructions (thus on particular instruction op codes).

To see the tracepoints currently in effect, issue the command:

```
DBG> SHOW TRACE
```

To cancel tracepoints at CALL or BRANCH instructions, use the CANCEL TRACE command with the appropriate qualifier, as follows:

```
DBG> CANCEL TRACE/CALL

DBG> CANCEL TRACE/BRANCH
```

The following example demonstrates the use of the /CALL and /BRANCH qualifiers in the SET, SHOW, and CANCEL TRACE commands:

```
DBG> SET TRACE/CALL                    !Trace call instructions.

DBG> SHOW TRACE                        !Show tracepoints in effect.
tracing /CALL instructions: CALLS, CALLG, BSBW, BSBB, JSB, RSB
and RET

DBG> GO 5660                           !Begin program execution at
start at PC 5660                       !address 5660.
trace at PC 5669: CALLS    #1,L^BAS$INIT_GOSUB
trace at PC 24566: JSB      @B^4(AP)
trace at PC 5457: CALLS    #1,L^BAS$PRINT
trace at PC 24918: CALLS    #4,W^24987
trace at PC 25003: JSB      L^8352
trace at PC 8407: JSB      L^94721
trace at PC 94877: RSB
trace at PC 8492: RSB
trace at PC 25190: JSB     L^34295
trace at PC 34333: RSB
          .
          .
          .
DBG> SHOW TRACE                        !Show tracepoints in effect.
tracing /CALL instructions: CALLS, CALLG, BSBW, BSBB, JSB, RSB
and RET

DBG> SET TRACE/BRANCH                  !Trace all branch
                                       !instructions.

DBG> SHOW TRACE                        !Show tracepoints in effect.
tracing /CALL instructions: CALLS, CALLG, BSBW, BSBB, JSB, RSB
and RET
tracing /BRANCH instructions: BNEQ, BEQL, BGTR, BLEQ, BGEQ, BLSS,
BGTRU,
        BLEQU, BVC, BVS, BGEQU, BLSSU, BRB, BRW, JMP, BBS, BBC,
BBSS, BBCS,
        BBSC, BBCC, BBSSI, BBCCI, BLBS, BLBC, ACBB, ACBW, ACBL,
ACBF, ACBD,
        ACBG, ACBH, AOBLEQ, AOBLSS, SOBGEQ, SOBGTR, CASEB, CASEW
and CASEL
```

```
DBG>GO 5444                                     !Restart program execution at
start at PC 5444:                               !address 5440.
trace at PC 5463: CALLS    #1,L^BAS$INIT_GOSUB
routine trace at PC BAS$INIT_GOSUB: JMP      L^24510
trace at PC 24552: BBC      #11,B^-26(R0),24559
trace at PC 24566: JSB      @B^4(AP)
trace at PC 5457: CALLS    #1,L^BAS$PRINT
routine trace at PC BAS$PRINT: JMP      L^24896
trace at PC 24906: BNEQ     24913
trace at PC 24911: BRB      24916
trace at PC 24918: CALLS    #4,W^24987
trace at PC 25003: JSB      L^8352
trace at PC 8359: BGTR     8366
trace at PC 8364: BGEQ     8377
trace at PC 8391: BEQL     8407
trace at PC 8407: JSB      L^94721
trace at PC 94728: BLBS     L^103336,94740
trace at PC 94752: BBCS     R2,L^104364,94762
trace at PC 94772: BEQL     94790
trace at PC 94804: BEQL     94817
trace at PC 94815: BRB      94829
trace at PC 94860: BLBC     B^4(SP),94869
trace at PC 94877: RSB
trace at PC 8413: CASEL    R0,#1,#2
                                     8431,
                                     8431,
                                     8458
trace at PC 8437: BNEQ     8444
          .
          .
          .
DBG> CANCEL TRACE/ALL                          !Cancel all tracepoints.

DBG> SHOW TRACE                                !Show tracepoints in effect.
%DEBUG-I-NOTRACES, no tracepoints are set, no opcode tracing
```

7.4 EXAMINING AND DEPOSITING DATA

This section discusses topics of particular interest in debugging at the assembly level. These are:

- Helpful techniques in examining and depositing instructions
- The replacing of instructions
- The examining and depositing of values in general purpose registers
- The evaluating of bit fields

For more information on examining and depositing data, see Chapter 4.

7.4.1 Techniques in Examining and Depositing Instructions

To examine and deposit instructions for the purposes of assembly-level debugging, you may find it helpful to issue the following commands:

- SET MODE NOSYMBOL -- When mode is nonsymbolic, symbolic operands within instructions are displayed as integers in the

current radix mode, and symbolic program locations are displayed as virtual addresses. Viewing program locations as a series of bytes in virtual memory is helpful because you can see how many bytes of memory are occupied by each instruction. Thus when you replace instructions, you can easily figure whether the new instruction is longer or shorter than the old one (see Section 7.4.2).

You can, of course, use symbolic mode to examine and deposit instructions in symbolic program locations. If you do so, you can specify the /NOSYMBOL command qualifier when you want to see what virtual memory address is represented by a program location symbol or what the value of a symbolic operand is.

- SET STEP INSTRUCTION -- When you set the step unit to instruction, you can step through your program by instruction rather than by line. This enables you to see an instruction with each single step and thus makes it easy to locate an instruction that needs modification.

  However, if the step parameter is set to LINE, you can override it with the /INSTRUCTION command qualifier when you want to step by instruction.

- SET TYPE INSTRUCTION -- When you set the type to instruction, the debugger displays program locations as instructions rather than as longword integers or some other type. This enables you to examine and deposit instructions in your program without always having to specify the /INSTRUCTION qualifier.

You must enclose instructions to be deposited in quotations (") or in apostrophes (').

7.4.2 Replacing Instructions

When you replace an instruction, you must ensure that the new instruction is the same length in bytes as the old instruction.

If the new instruction is longer than the old instruction, you cannot deposit it without overwriting, and thereby destroying, the subsequent instruction.

If the new instruction occupies fewer bytes of memory than the old instruction, it can be deposited without overwriting previous or subsequent instructions; however, you must deposit NOP instructions (instructions that cause "no operation") in bytes of memory left unoccupied after the replacement.

The debugger does not warn that an instruction you are depositing will overwrite a subsequent instruction, nor does the debugger remind you to fill in vacant bytes of memory with NOPs. Therefore, careful calculation of instruction length is required in replacing instructions.

The following example demonstrates the replacing of an instruction with an instruction of equal length:

```
DBG> SHOW STEP
step type: nosystem, by line, over routine calls, nosource
                                        !Display current
                                        !step parameters.
```

```
DBG> SET STEP INSTRUCTION                 !Set step unit to instruction.

DBG> SHOW MODE                            !Display current modes.
modes: symbolic, decimal

DBG> SET MODE NOSYMBOL                    !Set mode to nonsymbolic.

DBG> SHOW TYPE                            !Display current type.
type: long integer

DBG> SET TYPE INSTRUCTION                 !Set default type to
                                          !instruction.

DBG> STEP                                 !Step by instruction.
start at 1577
stepped to 1584: PUSHAL  (R11)

DBG> STEP                                 !Step by instruction.
start at 1584
stepped to 1586: CALLS    #1,L^2224 !Replace this instruction.

DBG> EXAMINE (RET)
1593:  CALLS   #0,L^2216                  !Subsequent instruction begins
                                          !at 1593.

DBG> EXAMINE 1584
1584:  PUSHAL  (R11)                      !Previous instruction begins
                                          !at 1584.

DBG> DEPOSIT/INSTRUCTION 1586 = "CALLS  #2,L^2224"
                                          !Deposit new instruction.

DBG> EXAMINE
1586:  CALLS   #2,L^2224                  !New instruction is
                                          !deposited.

DBG> EXAMINE (RET)
1593:  CALLS   #0,L^2216                  !Subsequent instruction is
                                          !unchanged.

DBG> EXAMINE 1584
1584:  PUSHAL  (R11)                      !Previous instruction is
                                          !unchanged, proof that new
                                          !instruction is of equal
                                          !length.
```

The following example demonstrates the replacing of an instruction with an instruction that occupies fewer bytes of memory (the same TYPE, MODE, and STEP parameters are in effect):

```
DBG> STEP                                 !This is the instruction
start at 1586                             !preceding the instruction
stepped to 1593: CALLS    #0,L^2216 !to be replaced.

DBG> STEP                                           !This is the
start at 1593                                       !instruction to
stepped to 1600: MOVF     #266027337,B^4(R11) !be replaced.

DBG> STEP                                       !This is the instruction
start at 1600                                   !following the instruct-
stepped to 1608: MULF3    (R11),(R11),R0 !ion to be replaced.
```

```
DBG> DEPOSIT 1600 = "MOVB  #18,B^4(R11) !Deposit new instruction
                                        !at address 1600.

DBG> EXAMINE .                          !Verify that the new
1600:  MOVB    #18,B^4(R11)             !instruction is deposited.

DBG> EXAMINE RET                        !Display the following
1604:  MFPR    #15,B^4(R11)             !instruction (the logical
                                        !successor). It is not the
                                        !subsequent instruction
                                        !examined above (at 1608).
                                        !Therefore, NOP instructions
                                        !must be deposited.

DBG> EXAMINE 1608                       !Verify that the subsequent
1608:  MULF3   (R11),(R11),R0           !instruction is intact.

DBG> EXAMINE 1600                       !Set current entity.
1600:  MOVB    #18,B^4(R11)

DBG> EXAMINE RET                        !Location 1604 is the first
1604:  MFPR    #15,B^4(R11)             !byte to be filled with
                                        !a NOP instruction.

DBG> DEPOSIT . = "NOP"                  !Deposit NOP into current
                                        !entity.

DBG> EXAMINE .                          !Verify the deposit.
1604:  NOP

DBG> EXAMINE RET                        !Locate next byte to be
1605:  REMQUE  B^4(R11),(R11)[R5]       !deposited into.

DBG> DEPOSIT . = "NOP"                  !Deposit NOP into current
                                        !entity.

DBG> EXAMINE .                          !Verify the deposit.
1605:  NOP

DBG> EXAMINE RET                        !Locate next byte to be
1606:  BICW3   #4,(R11)[R5],(R11)       !deposited into.

DBG> DEPOSIT . = "NOP"                  !Deposit NOP into current
                                        !entity.

DBG> EXAMINE .                          !Verify the deposit.
1606:  NOP

DBG> EXAMINE RET                        !Locate next byte to be
1607:  RET                              !deposited into.

DBG> DEPOSIT . = "NOP"                  !Deposit NOP into current
                                        !entity.

DBG> EXAMINE .                          !Verify the deposit.
1607:  NOP

DBG> EXAMINE RET                        !Subsequent instruction is
1608:  MULF3   (R11),(R11),R0           !the desired logical
                                        !successor.

     !Final check that sequence is o.k.
```

```
DBG> EXAMINE 1593                        !Verify that the previous
1593:  CALLS   #0,L^2216                 !instruction is intact.

DBG> EXAMINE <RET>                       !The logical successor is
1600:  MOVB    #18,B^4(R11)              !the new instruction.

DBG> EXAMINE <RET>
1604:  NOP

DBG> EXAMINE <RET>
1605:  NOP

DBG> EXAMINE <RET>
1606:  NOP

DBG> EXAMINE <RET>
1607:  NOP

DBG> EXAMINE <RET>                       !The subsequent instruction
1608:  MULF3   (R11),(R11),R0            !is intact.
```

7.4.3 Examining and Depositing Values in Registers

VAX-11 provides 16 general purpose registers, some of which are used for temporary address and data storage. You can examine the contents of any register by specifying that register in an EXAMINE command, and you can deposit values into any register by specifying that register as the address-expression in a DEPOSIT command.

The following symbols denote the 16 VAX-11 registers.

- The letter R followed by a numeral from 0 through 11 represents the corresponding VAX-11 general purpose register. Examples: R0, R1, R2, R3, R4, R5,...R11.
- PC represents the program counter.
- SP represents the stack pointer.
- AP represents the argument pointer.
- FP represents the frame pointer.

Note that together with the PSL, the symbols representing the VAX-11 registers are called debugger permanent symbols. Further, if language is set to VAX-11 PASCAL, you must prefix each of the debugger permanent symbols with a percent sign (%). (See Section 2.2.1 for more information on the debugger permanent symbols.)

The following example demonstrates how to examine and deposit values into the VAX-11 registers:

```
DBG> SHOW MODE                           !Display current mode.
modes: symbolic, decimal

DBG> SHOW TYPE                           !Display current type.
type: long integer

DBG> EXAMINE SP                          !Examine value of the stack
SP:  2147278720                          !pointer.
```

```
DBG> DEPOSIT SP = 33               !Deposit 33 into SP.

DBG> EXAMINE .                     !Check the value of SP.
SP:  33

DBG> EXAMINE R11                   !Examine contents of R11.
R11:  1024

DBG> DEPOSIT R11 = 444             !Deposit new value into R11.

DBG> EXAMINE R11                   !Check the value of R11.
R11:  444
```

7.4.3.1 **The Processor Status Longword (PSL)** - The PSL is a 32-bit VAX-11 register whose value represents a number of processor state variables. The first 16 bits of the PSL (referred to separately as the Processor Status Word, or PSW) contains unprivileged information about the current processor state; the values of these bits may be controlled by a user program. The latter 16 bits of the PSL, bits 16 through 31, contain privileged information and should not be altered by the user process.

To examine the contents of the PSL, issue the command:

```
DBG> EXAMINE PSL
```

In nonsymbolic mode, the debugger displays the value of the PSL in the current radix mode. In symbolic mode, the debugger displays the 32-bit PSL in the following format:

```
PSL:  CMP TP FPD IS CURMOD PRVMOD IPL DV FU IV T N Z V C
       n   n  n   n  mode   mode   lv  n  n  n n n n n
```

In this display, "n" may be 0 or 1; "mode" may be either KERN, EXEC, SUPR, or USER; and "lv," the interrupt priority level, may be a hexadecimal number from 0 through 1F.

The high word of the PSL (bits 16 through 31) are represented in the above formatted display, though they should not be altered by the user program. The following are the high word keys and their meanings (the low word keys are shown in Table 7-1):

- CMP -- compatibility mode
- TP -- trace pending
- FPD -- first part done
- IS -- interrupt stack
- CURMOD -- current mode
- PRVMOD -- previous mode
- IPL -- interrupt priority level

When you deposit into the PSL, your purpose is to enable or disable certain processor state conditions. Table 7-1 contains a list of the processor state conditions that you can manipulate. Column 1 of Table 7-1 lists the bits by number, from 0 to 15. Column 2 lists the corresponding key name, which is displayed in the symbolic representation of the PSW. Column 3 lists a corresponding key number, in hexadecimal radix, that is used in the DEPOSIT command to set the corresponding bit. Column 4 contains a description of each bit.

Table 7-1: PSL Modification Values

Bit	Key	Key Number (Hex)	Description
15		0	(Must be zero)
14		0	(Must be zero)
13		0	(Must be zero)
12		0	(Must be zero)
11		0	(Must be zero)
10		0	(Must be zero)
9		0	(Must be zero)
8		0	(Must be zero)
7	DV	80	Decimal overflow trap enable
6	FU	40	Floating underflow trap enable
5	IV	20	Integer overflow trap enable
4	T	10	Trace trap condition code
3	N	8	Negative condition code
2	Z	4	Zero condition code
1	V	2	Overflow condition code
0	C	1	Carry condition code

To deposit a value into the PSL, determine which bits you want set; add their corresponding key numbers together; then use the sum as the "expression" in the command:

```
DBG>DEPOSIT/WORD/HEX PSL = expression
```

NOTE

If you deposit into the high word of the PSL, bits 16 through 31, a reserved operand fault is generated when control is returned to your program by a STEP or GO command.

The following example demonstrates how to examine and modify the PSL:

```
DBG>SHOW MODE                        !Display current modes.
modes: symbolic, decimal

DBG>SHOW TYPE                        !Display current type.
type: long integer
```

```
DBG>DEPOSIT/WORD PSL = 0               !Disable all conditions in
                                       !PSL.  In other words,
                                       !clear bits 0 thru 15.

DBG>EXAMINE/SYMBOL PSL
PSL:    CMP TP FPD IS CURMOD PRVMOD IPL DV FU IV T N Z V C
        0   0  0   0  USER   USER   00  0  0  0  0 0 0 0 0

                                       !Display formatted PSL.
                                       !All bits are cleared.

DBG>DEPOSIT/WORD/HEX PSL = 80          !Key-number 80 from Table 7-1
                                       !enables the decimal overflow
                                       !trap; this is key "DV" in
                                       !the formatted display.

DBG>EXAMINE/SYMBOL PSL
PSL:    CMP TP FPD IS CURMOD PRVMOD IPL DV FU IV T N Z V C
        0   0  0   0  USER   USER   00  1  0  0  0 0 0 0 0

                                       !Verify that the
                                       !DV bit is set.
```

7.4.4 Evaluating Bit Fields

When the current language is set to VAX-11 MACRO, you can evaluate a single bit or a bit field in a memory location using the EVALUATE command.

Bits of the memory location are specified by number from high bit 31 to low bit 0. The specified bit field is enclosed in angle brackets ("<" and ">") with a colon (:) separating the numbered high and low bits in the field. If a single bit is to be evaluated, its number appears on both sides of the colon.

Note that in VAX-11 MACRO, the EVALUATE command displays the virtual address denoted by the address expression, not the contents of that address as is the case in high-level languages. Therefore, to evaluate a bit field that is part of a memory location, you must prefix the corresponding address expression with the "contents of" operator (@); otherwise the debugger evaluates the specified bit field of the address.

The following command shows the format used to display the value of a bit field residing within the memory location denoted by the address expression:

```
EVALUATE @address-expression<high-bit:low-bit>
```

As a result of this command, the debugger evaluates the specified bit field, including both the high and low bits, and returns the value in the current radix.

The following example demonstrates how to evaluate bit fields:

```
DBG> SET LANGUAGE MACRO                !Set the current language
                                       !to MACRO.  This is a
                                       !necessary step.

DBG> SH MODE                           !Display default modes.
modes: symbolic, hexadecimal

DBG> DEPOSIT J = 0ABCD                 !Deposit a hexadecimal value
                                       !into J.
```

```
DBG> EXAMINE J                     !Verify the deposit.
000002C8:  0000ABCD

DBG> E/BYTE J                      !Examine the first byte of J.
000002C8:  0CD

DBG> EVALUATE J                    !Determine the virtual address
000002C8                           !of J. (The EVALUATE command
                                   !in VAX-11 MACRO is equivalent
                                   !to the EVALUATE/ADDRESS
                                   !command in a high-level
                                   !language.)

DBG> EV @J<7:0>                    !Display the value of bits 7
000000CD                           !through 0 in J (the first
                                   !byte of J). Note the use of
                                   !the "contents of" operator
                                   !(@).

DBG> EV @J<3:0>                    !Display the value of the
0000000D                           !first four bits of J. Note
                                   !that four bits comprise a
                                   !hexadecimal digit.

DBG> EV @J<11:8>                   !Display the value of bits 11
0000000B                           !through 8.

DBG> EV @J<3:3>                    !Display the value of bit 3.
00000001                           !Note that bit 3 is set.

DBG> EV @J<1:1>                    !Display the value of bit 1.
00000000                           !Note that bit 1 is not set.

DBG> EV 2C8<3:0>                   !Display the value of bits 3
00000008                           !through 0 of the hexadecimal
                                   !address 2C8.

DBG> EV @2C8<3:0>                  !Display the value of bits 3
0000000D                           !through 0 of the contents of
                                   !hexadecimal address 2C8.

DBG> EV @2C8<15:12>                !Display the value of bits 15
0000000A                           !through 12 of the contents
                                   !of hexadecimal address 2C8.
```

7.5 DEBUGGING SHAREABLE IMAGES

A shareable image is an image that is not directly executable. To execute, a shareable image must be included as input in a linking operation that produces an executable image; then, when the executable image executes, the shareable image may also execute.

As far as debugging is concerned, the one or more routines that constitute a shareable image may exist in any of the following stages:

1. Routines that have been compiled but not linked; that is, one or more object modules
2. Routines that have been compiled and linked but not installed; that is, an uninstalled shareable image
3. Routines that have been compiled, linked, and installed; that is, an installed shareable image

A shareable image in any of the above stages may be included as input in a linking operation that produces an executable image. Therefore, when that executable image is run with debugger control, a shareable image in any of these stages can be debugged -- to some extent. The following sections discuss the debugging of shareable images in each of the above three stages.

For information on the linking of shareable images and on the image map, see the VAX-11 Linker Reference Manual. For information on the installing of shareable images, see the chapter on the Install Utility (INSTALL) in the VAX-11 Utilities Reference Manual.

7.5.1 Debugging Shareable Image Object Modules

To debug a shareable image that has not been linked, you include the object modules that make up (or will make up) the shareable image as input in the linking operation that produces the executable image. Next, you run the executable image with debugger control.

Providing you have specified the /DEBUG command qualifier at compile and link time, you have access to all the symbolic information in the shareable image modules. You can debug them in the same way that you debug any other modules in your program.

7.5.2 Debugging Uninstalled Shareable Images

To debug an uninstalled shareable image or an uninstalled copy of an installed shareable image, you include the shareable image as input in the linking operation that produces the executable image. Next, you run the executable image with debugger control.

You do not have access to any symbols within the shareable image. However, you do have access to the shareable image name and can, therefore, locate the starting address of the shareable image.

The name of every shareable image included in the linking operation is available to the debugger at run time. The debugger prefixes the file name of each shareable image with the name SHARE$ and recognizes this new name during a debugging session.

Thus, you need only prefix the shareable image name with SHARE$ and use this new name in the EVALUATE/ADDRESS command to determine the starting virtual address of the shareable image.

For example, assume that the VAX-11 Run-Time Library (VMSRTL.EXE), which is a shareable image, is included in the linking operation. You can determine the starting address of VMSRTL.EXE during the debugging session by issuing the command:

```
DBG>EVALUATE/ADDRESS SHARE$VMSRTL
```

Since you do not have access to symbol names within the shareable image, you must debug using virtual addresses. You can, for example, set breakpoints or watchpoints at locations within the shareable image by expressing these locations as byte offsets from the starting address of the shareable image. Likewise, you can examine and deposit data in program locations by expressing these locations as byte offsets from the starting address of the shareable image.

To debug the shareable image effectively in this manner, you must have a compiler listing of the shareable image. The compiler listing will enable you, among other things, to determine the offset in bytes, from the beginning of the routine, of instructions or variables of interest. To obtain a listing, you must specify the /LIST qualifier when you compile the shareable image routine(s).

In addition, you must have a full image map of the shareable image. Using this map, you can determine, among other things, the total length in bytes of the shareable image, as well as the layout in memory of the various portions of the now linked modules.

For example, using the image map, you can determine whether or not an access violation resulted from execution of code inside or outside that shareable image (since you know both the starting address of the shareable image and the length of the shareable image). If the error occurred in the shareable image, the image map and compiler listing also enable you to determine the exact location of the error within the shareable image.

To obtain a full image map, you must specify the /MAP and /FULL qualifiers when you link the object modules to create the shareable image.

7.5.3 Debugging Installed Shareable Images

The debugging of an installed copy of a shareable image is not recommended. First, it is probably not necessary since a shareable image is usually installed only after it has been tested and debugged. Second, since the image sections in the installed shareable image have become global sections and are meant to be shared by multiple processes, they will in most cases be protected from write access. You cannot, therefore, set breakpoints or alter the contents of program locations within the installed copy of the shareable image.

If you must debug an installed shareable image, you should create a private copy of the shareable image. Then, by using a logical name assignment, you should cause the private copy (rather than the installed copy) to be activated with the executable image. In this way, you can debug the shareable image in the manner described above for uninstalled shareable images.

In the following example, the DCL command COPY creates a private copy of the installed shareable image LBRSHR.EXE. Then, the DCL command DEFINE causes the private copy to be activated at image activation time:

```
$  COPY SYS$LIBRARY:LBRSHR.EXE MYDISK:[MYDIR]

$  DEFINE LBRSHR MYDISK:[MYDIR]LBRSHR

$  RUN PROG
```

CHAPTER 8

DISPLAYING SOURCE CODE

This chapter describes a new source-language display feature that is available -- in some languages only -- with Version 3.0 of VAX/VMS.

This feature makes it possible to display programming statements in the language in which the program was written. Such programming statements are referred to in this manual as source code.

The debugger identifies a line of source code (a source line) by the line number assigned to it by the compiler. The compiler assigns a line number to each line of source code in the program in sequential order from the first to the last line. Line numbers appear on a compiler listing, which the compiler generates when the /LIST command qualifier is specified at compile time.

The display of source lines is independent of program execution; that is, you may display source lines from any region of your program without such display affecting the value of the program counter (PC) or processor status longword (PSL).

You can display source code by specifying any of the following forms of input parameter:

- A compiler-assigned line number

- An address expression that denotes a program location that has a corresponding line number

- A source code string that identifies a source line by its appearance on that line

It is also possible to display source lines along with other debugger display such as that resulting from the execution of a STEP command, a breakpoint hit, or a watchpoint hit.

Section 8.1 describes how the debugger locates source files; it also describes how to direct the debugger to locate source files that have been moved to a different directory since being compiled. Sections 8.2 through 8.5 describe how to display source lines in each of the ways mentioned above. Section 8.6 describes how to establish parameters that govern the display of source lines.

8.1 LOCATION OF SOURCE FILES

A source file is a file containing statements in a programming language. A compiler processes source files to generate object modules. If the /DEBUG qualifier is specified at compile time, the compiler generates Source Line Correlation records for inclusion in the Debug Symbol Table (DST).

Source Line Correlation records contain the full file specification (that is, device name, directory name, file name, file type, and version number) of each source file that contributes to each object module, and they specify which source record in which source file corresponds to which line number in the object module. Source Line Correlation records are passed to the linker and made available to the debugger at run time.

During a debugging session, therefore, the debugger need only refer to its DST to locate, for any object module, the source file containing the source lines that you want to display.

However, if a source file has been moved to another directory since being compiled, the full file specification of the source file as listed in the DST record will no longer be correct, and the debugger will not be able to locate the source file using the DST record. In this case, you must direct the debugger to the current location of the source file by means of the SET SOURCE command.

8.1.1 SET, SHOW, and CANCEL SOURCE Commands

If you have moved a source file to another directory, issue the SET SOURCE command in the following format:

```
SET SOURCE[/MODULE=modname] dirname[,dirname...]
```

If you specify the optional /MODULE=modname command qualifier, the debugger looks in the specified directory or directories only when it is locating the source file(s) for the specified module.

If you issue a SET SOURCE command without the /MODULE=modname command qualifier, the debugger looks in the specified directory or directories when it is locating the source file(s) for any module not mentioned in a previous SET SOURCE/MODULE=modname command.

In sum, the SET SOURCE/MODULE command tells the debugger where to find source files for a particular module, whereas the SET SOURCE command tells the debugger where to find source files for modules that were not mentioned explicitly in SET SOURCE/MODULE commands.

The dirname parameter may consist of one, several, or all fields in a full file specification, though usually it is only a directory name. The following is the format of a full file specification:

```
node::device:[directory]file-name.file-type;version-number
```

When specifying any of these fields, you must include the punctuation for that field, as shown in the above format. For example, to specify the relocation of source files to the directory NEWDIR on the disk NEWDISK, issue the following command:

```
DBG> SET SOURCE NEWDISK:[NEWDIR]
```

The debugger processes the dirname parameter by inserting the specified field(s) in the file specification of the source file as it appears in the DST. In this way, the debugger creates a new file specification, which it then uses to locate the source file.

When a source file is moved to another directory, the version number of the source file may change. Hence, to locate the correct version of the source file in the event that a version number was not specified in the dirname parameter, the debugger inserts the match-all

wild card character (*) in the version number field of the new file specification. As a result, all versions of the moved source file are searched until the correct version is located. The correct version of the source file is the version that has the same revision date and time, the same file size, the same record format, and the same file organization as the original compile-time source file.

You can specify more than one dirname parameter in a single SET SOURCE command by separating each dirname parameter with a comma (,). In this case, the debugger constructs a new file specification for each dirname parameter specified and uses this list of file specifications to locate source files for the object module. Since the dirname parameter is most often a directory name, when you specify more than one dirname parameter in a single SET SOURCE command, you establish a source directory search list.

When a source directory search list has been established for a module or modules, the debugger locates the source file(s) for the designated module(s) by searching the first directory on the list, then the next, and so on, until it either locates the source file or exhausts the list. Using a source directory search list is particularly helpful when relocated source files are in several directories.

The SHOW SOURCE command displays the source directory search lists currently in effect. The following is the format of the SHOW SOURCE command:

```
SHOW SOURCE
```

If a source directory search list has been established for all modules, the SHOW SOURCE command indicates the name(s) of each directory specified and indicates that the list applies to all modules.

If a source directory search list has been established for one or more modules, the SHOW SOURCE command displays the name(s) of each directory specified and displays the name(s) of the module(s) to which the directory search list applies.

If no source directory search list has been established, the SHOW SOURCE command indicates that no such list is currently in effect.

The CANCEL SOURCE command cancels the effects of previous SET SOURCE commands. The following is the format of the CANCEL SOURCE command:

```
CANCEL SOURCE [/MODULE=modname]
```

The CANCEL SOURCE command, without the /MODULE=modname command qualifier, cancels the effect of a previous SET SOURCE command, but does not cancel the effect of any previous SET SOURCE/MODULE=modname command.

The CANCEL SOURCE/MODULE=modname command cancels the effect of a previous SET SOURCE/MODULE=modname command in which the same module name was specified; it does not cancel the effect of a previous SET SOURCE command or of a SET SOURCE/MODULE=modname command in which a different module name was specified.

In all cases, when a source directory search list has been canceled, the debugger again expects the source file(s) corresponding to the designated modules to be in the same directories they were in at compile time.

8.1.2 Example

Assume that you write a VAX-11 COBOL program with the file name TEST.COB;8 and that this program contains two modules MODA and MODB. Assume further that your default disk is DB4 and that, when you compile this program, your default is set to [ME.COBPROG], a subdirectory in which you always keep COBOL programs. At compile time, then, your source file has the full file specification:

```
DB4:[ME.COBPROG]TEST.COB;8
```

Assume that you link your program with a VAX-11 FORTRAN program, consisting of a single module CHECK, and that the full file specification of this program at compile time is:

```
DB4:[ME.FORPROG]CHECK.FOR;2
```

When you run the program and want to display source lines, the debugger looks for the source lines of modules MODA and MODB in DB4:[ME.COBPROG]TEST.COB;8 and for the source lines of module CHECK in DB4:[ME.FORPROG]CHECK.FOR;2.

Assume that you move TEST.COB to a new subdirectory [ME.TEST]. Now when you run the program and request the debugger to display source lines from MODA, the debugger looks in DB4:[ME.COBPROG]TEST.COB;8 but does not find the source file.

To direct the debugger to the correct directories, you can establish a source directory search list by issuing the following command:

```
DBG> SET SOURCE [ME.TEST],[ME.FORPROG]
```

The debugger processes this command by substituting the directory name [ME.TEST] in the compile-time file specifications of the source files corresponding to MODA, MODB, and CHECK. As a result, the debugger would successfully locate the source files for modules MODA and MODB (since they were moved to this directory) but would not locate the source file for module CHECK (since it was not moved from its compile-time directory [ME.FORPROG]). However, after failing to find the source file for module CHECK in the directory [ME.TEST], the debugger would substitute the next directory in the source directory search list ([ME.FORPROG]) and would then successfully locate the source file there.

Another way to direct the debugger to the correct directories is to issue the following two commands:

```
DBG> SET SOURCE/MODULE=MODA [ME.TEST]
DBG> SET SOURCE/MODULE=MODB [ME.TEST]
```

The debugger processes these commands by inserting the directory name [ME.TEST] in the compile-time file specifications of the source files corresponding to MODA and MODB. Now whenever you request a display of source lines from MODA or MODB, the debugger locates the source file in DB4:[ME.TEST]TEST.COB.

In this example, it is necessary to issue two SET SOURCE/MODULE=modname commands, rather than a single SET SOURCE command, because the program contains another module CHECK whose corresponding source file was not moved to the directory [ME.TEST]. As a result, the debugger continues to look in the original compile-time directory DB4:[ME.FORPROG]CHECK.FOR;2 when displaying source lines for module CHECK.

8.2 DISPLAY BY LINE NUMBER

The TYPE command allows you to display one or more source lines by specifying one or more compiler-assigned line numbers, where each line number designates a line of source code.

The following is the format of the TYPE command:

```
TYPE [ [modname\]line-number[:line-number] -
     [,[modname\]line-number[:line-number]...] ]
```

If you specify a single line number, the debugger displays the source line corresponding to that line number.

If you specify a list of line numbers, separating each with a comma (,), the debugger displays the source line corresponding to each of the line numbers.

If you specify a range of line numbers, separating the starting and ending line numbers in the range with a colon (:), the debugger displays the source lines corresponding to that range of line numbers.

You can read through all the source language statements in your program by specifying a range of line numbers starting from 1 and ending at a number equal to or greater than the largest line number in the program listing.

After displaying a source line, you can display the next line by issuing a TYPE command without a line number, that is, by issuing a TYPE command and then pressing the RETURN key. You can then display the next line and successive lines by repeating this sequence, in this way reading through your source program one line at a time.

You can specify a module name with the line number(s) to indicate that the line(s) are located in that module. In this case, you enter the module name, a backslash (\), and the line number(s), without intervening spaces.

If you do not specify a module name with the line number(s), the debugger uses the current scope setting to determine which module to use. In this case, the current scope is either the first module designated in a SET SCOPE command or, if a SET SCOPE command was not issued, the module containing the current PC.

The following example demonstrates how to use the TYPE command:

```
DBG>TYPE 160                          !Display the source line
module COBOLTEST                      !designated by line
   160: START-IT-PARA.                !number 160.

DBG>TYPE RET                          !Display the source line
module COBOLTEST                      !following the last
   161:          MOVE SC1 TO ES0.     !displayed source line.

DBG>T 160:163                         !Display the range of source
module COBOLTEST                      !lines corresponding to the
   160: START-IT-PARA.                !range of specified line
   161:          MOVE SC1 TO ES0.     !numbers.
   162:          DISPLAY ES0.
   163:          MOVE SC1 TO ES1.
```

```
DBG>TYPE COBOLTEST\160,22:24          !Display a single line
module COBOLTEST                      !and a range of lines of
   160: START-IT-PARA.                !source code in a specified
module COBOLTEST                      !module.
    22: 02       SC2V2   PIC S99V99       COMP VALUE  22.33.
    23: 02       SC2V2N  PIC S99V99       COMP VALUE -22.33.
    24: 02       CPP2    PIC PP99         COMP VALUE  0.0012.
```

8.3 DISPLAY BY ADDRESS EXPRESSION

By specifying the /SOURCE qualifier in the EXAMINE command, you can display the source line(s) corresponding to the location(s) designated by one or more address expressions.

The debugger evaluates each address expression to derive a virtual address, determines which compiler-assigned line number corresponds to each virtual address, and then displays the source line(s) designated by the line number(s).

The following is the format of the EXAMINE command:

```
EXAMINE[/qualifier[/qualifier]] -
        [address-expression[:address-expression]-
        [,address-expression[:address-expression]...]]
```

The format of the EXAMINE command allows you to specify:

- A single address-expression parameter
- A list of address-expression parameters
- A range of address-expression parameters
- A list of ranges of address-expression parameters

In each case, when you specify the /SOURCE qualifier, the debugger evaluates any specified address expression to determine its corresponding source line.

If you specify a single address expression, the debugger evaluates the address expression to derive a virtual address, determines what line number corresponds to that virtual address, and then (as in the TYPE command) displays the source line designated by that line number.

If you specify more than one address expression (that is, a list), with a comma (,) separating each address expression, the debugger evaluates each address expression as described above and displays, for each specified address expression, a corresponding source line.

If you specify two address expressions, separated by a colon (:), the debugger evaluates each address expression to derive a range of virtual addresses, determines what line number(s) correspond to the range of addresses, and then displays the source line(s) designated by the line number(s). In this case, both addresses in the range must correspond to line numbers in the same module, and the first of the two line numbers must be less than or equal to the second.

The following example demonstrates how to use the EXAMINE/SOURCE command:

```
DBG>EX/SOURCE %PC
!%DEBUG-W-NOSRCLIN, no source line for address 7FFF005C
                                             !The address expression %PC
                                             !designates the address at
                                             !which the contents of the PC
                                             !is stored.  There is no
                                             !source line corresponding to
                                             !this address.  To get the
                                             !desired result, you must
                                             !specify the "contents of"
                                             !operator, the period (.),
                                             !as shown in the following
                                             !command.

DBG>EX/SOURCE .%PC                           !This command displays the
module COBOLTEST                             !source line corresponding
    162:          DISPLAY ES0.               !to the line containing
                                             !the PC.  In effect, it
                                             !displays the source line
                                             !currently being executed.

DBG>EX/SOU 2150                              !This command displays
module COBOLTEST                             !the source line corresponding
    165:          MOVE SC1 TO ES2.           !to the virtual address 2150.

DBG>EX/SOURCE 2150:2200                      !This command displays the
module COBOLTEST                             !range of lines of source
    165:          MOVE SC1 TO ES2.           !code that correspond
    166:          DISPLAY ES2.               !to the range of specified
    167:          MOVE SC2V2 TO ES0.         !virtual addresses.
    168:          DISPLAY ES0.
    169:          MOVE SC2V2 TO ES1.

DBG>EX/SOURCE 2100:2120,2150:2175            !This command displays the
module COBOLTEST                             !list of ranges of source
    161:          MOVE SC1 TO ES0.           !lines corresponding to
    162:          DISPLAY ES0.               !the list of ranges of
    163:          MOVE SC1 TO ES1.           !specified virtual addresses.
module COBOLTEST
    165:          MOVE SC1 TO ES2.
    166:          DISPLAY ES2.
    167:          MOVE SC2V2 TO ES0.
```

8.4 DISPLAY DURING PROGRAM EXECUTION

When the SOURCE parameter is in effect by virtue of a SET STEP SOURCE command, the debugger displays source lines when any one of the following occurs:

- You issue a STEP command and do not specify the /NOSOURCE qualifier. In this case, the debugger displays the source line following the last line or instruction executed.
- A breakpoint is activated. In this case, the debugger displays the source line at the breakpoint location.
- A watchpoint is activated. In this case, the debugger displays the source line corresponding to the instruction that caused the watchpoint activation.

When the NOSOURCE parameter is in effect by virtue of a SET STEP NOSOURCE command, the debugger does not display source lines when STEP commands are executed or when breakpoints or watchpoints are activated. NOSOURCE is the default parameter.

When you specify either the /SOURCE or /NOSOURCE qualifiers in the STEP command, they override, for the duration of that STEP command, any specified or default source parameter currently in effect. However, as command qualifiers, /SOURCE and /NOSOURCE do not affect whether or not the debugger displays source lines upon breakpoint or watchpoint activation; they only affect whether or not the debugger displays source lines when STEP commands are executed.

Thus, when the SOURCE parameter is in effect, specifying the /NOSOURCE command qualifier in the STEP command suppresses the display of source line(s) for the duration of that STEP command. The debugger continues to display source lines when breakpoints or watchpoints are activated or when subsequent STEP commands are executed (unless the /NOSOURCE command qualifier is again specified).

Also, when the NOSOURCE parameter is in effect (whether by default or by an explicit SET STEP NOSOURCE command), specifying the /SOURCE command qualifier in the STEP command causes the debugger to display source line(s) for the duration of that STEP command. The debugger does not display source lines when breakpoints or watchpoints are activated or when subsequent STEP commands are executed (unless the /SOURCE command qualifier is again specified).

The following demonstrates how to display lines of source code during program execution:

```
DBG> SET STEP SOURCE                       !Request source line display
                                           !when a STEP command is
                                           !executed or when a
                                           !breakpoint or watchpoint
                                           !is activated.

DBG> STEP                                  !Execute a line of code.
routine start at COBOLTEST                 !Since the SOURCE parameter
stepped to COBOLTEST\START-IT              !is in effect, the debugger
   161:        MOVE SC1 TO ES0.            !displays the source line.

DBG> SET BREAK %LINE 163                   !Set a breakpoint at
                                           !line number 163.

DBG> GO
start at COBOLTEST\START-IT
break at COBOLTEST\START-IT\START-IT-PARA\%LINE 163
   163:        MOVE SC1 TO ES1.
                                           !Begin execution at the
                                           !current location.  Source
                                           !line display occurs when
                                           !the breakpoint is reached.

DBG> SET BREAK %LINE 165                   !Set a breakpoint at
                                           !line number 165.

DBG> SET STEP NOSOURCE                     !Specify that source lines
                                           !not be displayed when STEP
                                           !commands are executed or
                                           !when breakpoints or
                                           !watchpoints are activated.
```

```
DBG> GO
start at COBOLTEST\%LINE 163
break at COBOLTEST\START-IT\START-IT-PARA\%LINE 165
                                        !Begin execution at the
                                        !current location.  Note that
                                        !a source line is not
                                        !displayed at breakpoint
                                        !activation because NOSOURCE
                                        !is in effect.

DBG> STEP/SOURCE                        !Execute a line of code and
start at COBOLTEST\%LINE 165            !display the source line
stepped to COBOLTEST\%LINE 166          !following the line executed.
   166:         DISPLAY ES2.            !The NOSOURCE parameter is
                                        !overriden for the duration
                                        !of the STEP command.

DBG> STEP                               !Execute a line of code.
start at COBOLTEST\%LINE 166            !Note that no source code
stepped to COBOLTEST\%LINE 167          !is displayed because the
                                        !SET STEP NOSOURCE command
                                        !was issued and the /SOURCE
                                        !qualifier was not specified
                                        !in the STEP command.
```

8.5 DISPLAY BY SEARCH STRING

The SEARCH command directs the debugger to search the source code for a specified string and to display the source line or lines containing an occurrence of the string.

The format of the SEARCH command follows:

```
SEARCH [/qualifier[/qualifier]] range string
```

The range parameter limits the debugger's search for occurrences of the string to specified regions of the source code. These regions may be specified in any of the following formats:

- MODNAME indicates a search of the specified module from line number 0 to the end of the module.
- MODNAME\LINE-NUM indicates a search of the specified module from the specified line number to the end of the module.
- MODNAME\LINE-NUM:LINE-NUM indicates a search of the specified module beginning at the line number specified to the left of the colon and ending at the line number specified to the right of the colon.
- LINE-NUM indicates a search of the module designated by the current scope setting from the specified line number to the end of the module.
- LINE-NUM:LINE-NUM indicates a search of the module designated by the current scope setting beginning at the line number specified to the left of the colon and ending at the line number to the right of the colon.
- NULL (that is, no entry) indicates a search of the same module as that from which a source line was most recently displayed (as a result of a SEARCH, TYPE, or EXAMINE/SOURCE command), beginning at the first line following the line most recently displayed and continuing to the end of the module.

The string parameter specifies the source code characters for which to search. If the string parameter is not specified, the debugger uses the last specified search string, that is, the string parameter specified in the last SEARCH command. If there was no previous search string, an error message results. The string parameter may be enclosed in quotation marks (") or in apostrophes (') or may be specified without delimiters.

If the string contains a quotation mark and you want to delimit the string with quotation marks, use a double quotation mark ("") to designate the quotation mark that is part of the string. Likewise, if the string contains an apostrophe and you want to delimit the string with apostrophes, use a double apostrophe ('') to designate the apostrophe that is part of the string.

A delimited string may contain spaces, tabs, or special characters. Specifying an undelimited string is more convenient because it saves keystrokes, but an undelimited string is subject to certain restrictions.

The following restrictions apply to an undelimited string:

- It must have no leading or trailing blanks or tabs.
- It must have no embedded semicolon (;).
- The range parameter must not be null; that is, an explicit range must be specified.

The /ALL command qualifier directs the debugger to search for all occurrences of the string in the specified range and display every line containing an occurrence of the string.

The /NEXT command qualifier directs the debugger to search for the first occurrence of the string in the specified range and to display only the line containing this occurrence. Subsequent lines in the specified range that contain the search string are not displayed. /NEXT is the default.

The /IDENTIFIER command qualifier directs the debugger to search for an occurrence of the string in the specified range but to display the string only if it is bounded on either side by a character that cannot be part of an identifier in the current language. (An identifier is the name associated with a data or program entity.)

Specifying /IDENTIFIER is useful when you are searching for an identifier but want to eliminate extraneous occurrences of the character(s) composing that identifier. For example, suppose you want to display all occurrences of the identifier X, but your program also contains identifiers XT and EXP. Obviously, you do not want every occurrence of XT and EXP to be displayed just because it contains the character X. So you specify the /IDENTIFIER command qualifier to direct the debugger to disregard occurrences of XT and EXP. The debugger disregards them because the X in both XT and EXP is bounded on at least one side by a character (a letter of the alphabet) that can be part of an identifier in the current language.

The /STRING command qualifier directs the debugger to search for and display the string as specified, and not to interpret the context surrounding an occurrence of the string, as it does in the case of /IDENTIFIER. /STRING is the default.

The SET SEARCH command establishes current SEARCH parameters for the debugger to use whenever a SEARCH command qualifier is not specified in a SEARCH command. The following is the format of the SET SEARCH command:

```
SET SEARCH parameter[,parameter]
```

SEARCH parameters determine whether the debugger searches for all occurrences (ALL) of the string or only the next occurrence (NEXT) of the string, and whether the debugger displays any occurrence of the string (STRING) or only those occurrences in which the string is not bounded on either side by a character that can be part of an identifier in the current language (IDENTIFIER).

If you do not specify SEARCH parameters with the SET SEARCH command, the debugger uses the default values NEXT and STRING.

You can override current SEARCH parameters for the duration of a single SEARCH command by specifying other SEARCH parameters as command qualifiers in the SEARCH command.

You can specify more than one SEARCH parameter in a single SET SEARCH by separating parameters with a comma.

The SHOW SEARCH command displays the current SEARCH parameters.

The following example demonstrates how to use the SEARCH, SET SEARCH, and SHOW SEARCH commands:

```
DBG> SHOW SEARCH
search settings: search for next occurrence, as a string
                                   !Display current SEARCH
                                   !parameters.

DBG> SEARCH/STRING/ALL 40:50 D
module COBOLTEST
    40: 02      D2N     COMP-2 VALUE -234560000000.
    41: 02      D       COMP-2 VALUE  222222.33.
    42: 02      DN      COMP-2 VALUE -222222.333333.
    47: 02      DR0     COMP-2 VALUE  0.1.
    48: 02      DR5     COMP-2 VALUE  0.000001.
    49: 02      DR10    COMP-2 VALUE  0.00000000001.
    50: 02      DR15    COMP-2 VALUE  0.0000000000000001.
                                   !Display all source lines
                                   !in the range of line numbers
                                   !40 to 50 that contain an
                                   !occurrence of the letter D.

DBG> SEARCH/IDENTIFIER/ALL 40:50 D
module COBOLTEST
    41: 02      D       COMP-2 VALUE  222222.33.
                                   !Display a line containing D
                                   !only if D is bounded on
                                   !both sides by a character
                                   !that cannot be part of an
                                   !identifier in the current
                                   !language.

DBG> SEARCH/NEXT 40:50 D
module COBOLTEST
    40: 02      D2N     COMP-2 VALUE -234560000000.
                                   !Display the first occurrence
                                   !of D in the range.
```

```
DBG> SEARCH/NEXT
module COBOLTEST
    41: 02        D         COMP-2 VALUE  222222.33.
                                         !Display the first occurrence
                                         !of D (the previous search
                                         !string) in the range begin-
                                         !ning at the line following
                                         !the last line displayed
                                         !(line 40 in the above com-
                                         !mand) and ending at the end
                                         !of the current module.

DBG> SEA 80:90
module COBOLTEST
    80: 02        LS10      PIC S9(10)      LEADING SEPARATE VALUE
     -: 1234567890.
                                         !Display all occurrences of D
                                         !(the previous search string)
                                         !in the range 80:90.  Note
                                         !that the line wraps.

DBG> SEA COBOLTEST\170 'D'
module COBOLTEST
   170:           DISPLAY ES1.
                                         !Display the first occurrence
                                         !of D in the module COBOLTEST
                                         !in the range beginning at
                                         !line 170 and ending at the
                                         !end of the module.

DBG> SEA COBOLTEST\150 ' E'
module COBOLTEST
   161:           MOVE SC1 TO ES0.
                                         !Display the first occurrence
                                         !of the delimited string in
                                         !the module COBOLTEST in the
                                         !range beginning at line 150
                                         !and ending at the end of
                                         !the module.

DBG> SEARCH/NEXT 1 ". "
module COBOLTEST
     2: PROGRAM-ID. COBOLTEST.
                                         !Display the first occurrence
                                         !of the delimited string in
                                         !the module COBOLTEST in the
                                         !range beginning at line 1
                                         !and ending at the end of
                                         !the module.

DBG> SET SEARCH IDENT                    !Set the current SEARCH
                                         !parameter to IDENTIFIER.

DBG> SHOW SEARCH
search settings: search for next occurrence, as an identifier
                                         !Display current SEARCH
                                         !parameters

DBG> SET SE AL                           !Set the current SEARCH
                                         !parameter to ALL.

DBG> SH SEA
search settings: search for all occurences, as an identifier
                                         !Display current SEARCH
                                         !parameters.
```

8.6 SOURCE DISPLAY PARAMETERS

This section describes parameters to set margins on the display of source code and to limit the number of source files that the debugger may keep open at any one time.

8.6.1 Margin Parameters

The SET MARGIN command specifies the leftmost source-line character position at which to begin display of a source line (the left margin) and/or the rightmost character position at which to end display of a source line.

By default, the debugger displays a source line beginning at character position 1 of the source line. Source-line character position 1 is actually character position 9 on your terminal screen. The first 8 character positions on the terminal screen are occupied by the line number and cannot be manipulated by the SET MARGIN command.

Increasing the left margin setting (from its default value of 1) is particularly useful when the source code is deeply indented; by eliminating the display of empty space, more space is available on the terminal line for the display of source code.

Decreasing the right margin setting from its default value of 255 prevents wrapping of long lines by truncating them. Since a wrapped line of source code requires two lines on the terminal, eliminating wrapping allows more lines of source code to be displayed on the terminal screen.

The SET MARGIN command affects only the display of source lines, that is, the display resulting from commands such as TYPE and EXAMINE/SOURCE. The SET MARGIN command does not affect the display resulting from commands (such as EXAMINE, EVALUATE, SHOW MODE, and so on) that do not display source code. If a command displays source code together with other information -- as, for example, STEP/SOURCE does -- the display of source code is affected by the current margin settings but the other information is not.

The following is the format of the SET MARGIN command:

```
             ( rm      )
SET MARGIN   ) lm:rm   (
             ) lm:     (
             (    :rm  )
```

If you specify a single number, the debugger sets the left margin to 1 and the right margin to the number specified.

If you specify two numbers, separated with a colon (:), the debugger sets the left margin to the number specified to the left of the colon and the right margin to the number specified to the right of the colon.

If you specify a single number followed by a colon, the debugger sets the left margin to the number specified and leaves the right margin unchanged.

If you specify a colon followed by a single number, the debugger sets the right margin to the number specified and leaves the left margin unchanged.

The SHOW MARGIN command displays the current margin settings for the display of source lines.

The following example demonstrates how to use the SET MARGIN and SHOW MARGIN commands:

```
DBG> SHOW MARGIN
left margin: 1 , right margin: 255
                                          !Display current margin
                                          !settings.

DBG> TYPE 116
module COBOLTEST
  116: 02       RA26    PIC A(26)      JUST RIGHT VALUE
    -: "ABCDEFGHIJKLMNOPQRSTUVWXYZ".
                                          !Display source line 116.
                                          !Note that the line wraps,
                                          !thus requiring two terminal
                                          !lines for its display.

DBG> SET MARGIN 50                        !Set the left margin to 1
                                          !and the right margin to 50.

DBG> SHOW MARGIN
left margin: 1 , right margin: 50
                                          !Display current margin
                                          !settings.

DBG> TY 116
module COBOLTEST
  116: 02       RA26    PIC A(26)      JUST RIGHT VALUE
                                          !With a right margin of 50,
                                          !characters on source line 116
                                          !at positions greater than 50
                                          !are not displayed.  As a
                                          !result, line 116 does not
                                          !wrap.

DBG> SET MARGIN 10:60
                                          !Set left margin to 10 and
                                          !right margin to 60.

DBG> TY 116
module COBOLTEST
  116: A26    PIC A(26)       JUST RIGHT VALUE  "ABCDEFGHI
                                          !Characters on source line 116
                                          !at positions less than 10 or
                                          !greater than 60 are not
                                          !displayed.

DBG> SET MARGIN :100                      !Set right margin to 100 and
                                          !leave left margin unchanged.

DBG> SHOW MARGIN
left margin: 10 , right margin: 100
                                          !Display current margin
                                          !settings.

DBG> SET MARGIN 5:                        !Set left margin to 5 and
                                          !leave right margin unchanged.

DBG> SHOW MARGIN
left margin: 5 , right margin: 100
                                          !Display current margin
                                          !settings.
```

8.6.2 Maximum Source Files Parameter

The SET MAX_SOURCE_FILES command specifies the maximum number of source files that the debugger may keep open at any one time.

The following is the format of the SET MAX_SOURCE_FILES command:

```
SET MAX_SOURCE_FILES n
```

The parameter n is a decimal integer whose value designates the maximum number of source files that the debugger may keep open at any one time. The value of n may not exceed 20. The default value is 5.

Opening a source file requires the use of an I/O channel, which is a limited system resource. Both the program and the debugger use I/O channels. To ensure that the debugger does not use all available I/O channels and thus cause the user program to fail (for lack of an available I/O channel), you can issue the SET MAX_SOURCE_FILES command to specify the maximum number of source files (and thus source file I/O channels) that the debugger may use at any one time.

Note that the value of MAX_SOURCE_FILES does not limit the number of source files that the debugger can open; rather, it limits the number that may be kept open at any one time. Thus, if the debugger reaches this limit, it must close a file in order to open another one.

Note too that setting MAX_SOURCE_FILES to a very small number can make the debugger's use of source files inefficient.

The SHOW MAX_SOURCE_FILES command displays the number of source files that the debugger may keep open at any one time.

The following example demonstrates how to use the SET MAX_SOURCE_FILES and SHOW MAX_SOURCE_FILES commands:

```
DBG> SHOW MAX_SOURCE_FILES          !Display the number of source
max_source_files: 5                 !files that the debugger may
                                    !keep open at the present
                                    !time.

DBG> SET MAX_SOURCE_FILES 8         !Set the number of source
                                    !files that the debugger may
                                    !keep open at any one time
                                    !to 8.

DBG> SHOW MAX_SOURCE_FILES          !Verify the change.
max_source_files: 8
```

8.7 DIFFERENCES BETWEEN SOURCE AND OBJECT CODE DUE TO OPTIMIZATION

When debugging with source code, you should keep in mind that, in general, there is no precise one-to-one correspondence between your source code and the compiler-generated object code.

Most compilers optimize the object code they produce so that the program will execute faster. One method of optimizing object code is to perform operations in a sequence different from the sequence specified in the source code.

As a result, the source code displayed by the debugger will not correspond exactly to the actual object code being executed. This is important to keep in mind, especially when using the SET STEP SOURCE, STEP/SOURCE, and EXAMINE/SOURCE commands.

To illustrate, the following example depicts a segment of source code from a FORTRAN program as it would appear on a compiler listing. This code segment sets the first ten elements of array A to the value 1/X.

```
line         source code
----         -----------
  5            DO 100 I=1,10
  6            A(I) = 1/X
  7        100 CONTINUE
```

As the compiler processes the source program, it determines that the reciprocal of X need only be computed once, not ten times as the source code specifies, since the value of X never changes in the DO-loop. The compiler thus generates optimized object code equivalent to the following code segment:

```
line         object code equivalent
----         ----------------------
  5            T = 1/X
               DO 100 I=1,10
  6            A(I) = T
  7        100 CONTINUE
```

In the optimized object code, the value of 1/X is computed once, saved in a temporary location, and then assigned to each A(I). The object code now executes faster, but it no longer corresponds exactly to the source code.

In this example, if you step to line 5 by issuing STEP/SOURCE (or SET STEP SOURCE followed by STEP), the debugger displays the source line as it appears in the source file, not the optimized object code equivalent that it is actually executing.

```
stepped to PROG_\%LINE 5
     5:        DO 100 I=1,10
```

At this point, if you issue another STEP command to execute line 5, the debugger executes line 5 of the optimized object code, not line 5 of the displayed source code. Thus, the program computes the reciprocal of X and sets up the DO loop, whereas the source display indicates only that the DO loop is set up.

This discrepancy is not obvious from looking at the displayed source line. Furthermore, if the computation of 1/X were to fail because X is zero, it would appear from inspecting the source display that a division by zero had occurred on a source line that contains no division at all.

This kind of apparent mismatch between source code and object code should be expected from time to time when debugging optimized programs. It can be caused not only by "code motions" out of loops, as in the above example, but by a number of other optimization methods as well. Optimization can cause segments of object code to move long distances from their original source code locations.

If you should encounter a program segment where source and object code do not seem to match, you can inspect the object code itself by using the EXAMINE/INSTRUCTION or STEP/INSTRUCTION commands. Alternatively, you can inspect a compiler-generated machine code listing. Using either of these methods, you should be able to determine what is happening at the object code level and thereby resolve the discrepancy between source line display and program behavior.

In addition, most languages allow you to specify the /NOOPTIMIZE command qualifier at compile time. Specifying the /NOOPTIMIZE qualifier inhibits compiler optimization, thereby eliminating discrepancies between source code and object code caused by optimization.

CHAPTER 9

DEBUGGER COMMANDS

This chapter contains an alphabetical list of the debugger commands. The following information is provided for each command:

- Command description
- Command format
- Command parameters
- Command qualifiers
- One or more examples

Acceptable abbreviations of debugger commands are used in the examples. Refer to Table 1-1 for a list of all debugger commands and their acceptable abbreviations.

@ file-spec

Description

Executes debugger commands contained in the specified file. A file specified in the @file-spec command is called a command procedure or a command file.

The command procedure can contain any debugger command, including another @file-spec command. When the debugger executes an EXIT command in a command procedure or reaches the end of the file, it returns control to the command stream that invoked the command procedure. A command stream can be the terminal, a previous command procedure, or a DO command sequence in a SET BREAK command.

If you enter SET OUTPUT VERIFY, all commands read from a command procedure are echoed on the terminal.

Format

@file-spec

Command Parameters

file-spec

Specifies the command procedure to be executed. If file-spec does not include a file type, the default file type COM is used. Note that file-spec is not delimited by quotation marks (") or apostrophes (').

Command Qualifiers

None.

Example

```
DBG> SET OUTPUT VERIFY      !Request echoing of commands
                            !before execution.

DBG> @MAIN
%DEBUG-I-VERIFYICF, entering indirect command file "MAIN"
SET MODU/ALL
SET BR SUB1
GO
routine start at MAIN\MAIN
routine break at SUB1\SUB1
E/WORD SUB1
SUB1\SUB1:  4800
E/I
SUB1\SUB1+02:  MOVAL  L^SUB1\Y,R11
EXIT
%DEBUG-I-VERIFYICF, exiting indirect command file "MAIN"
DBG>
```

CALL

Description

Calls the specified procedure. You can specify the procedure's name or virtual address. If the procedure requires one or more arguments, you must specify them in the CALL command.

When you issue a CALL command, the debugger takes the following action:

1. Saves the current values of the general registers
2. Constructs an argument list
3. Executes a call to the routine specified in the command and passes any arguments
4. Executes the routine
5. Displays the value returned by the routine in R0
6. Restores the values of the general registers to the values they had just previous to the CALL command
7. Issues its prompt

The debugger assumes that the called procedure conforms to the VAX-11 procedure calling standard (see the VAX Architecture Handbook).

You can call a procedure and debug it independently of the rest of the program.

Format

```
CALL routine-name [(argument[,argument...])]
```

Command Parameters

routine-name

Specifies the name or the virtual address of the procedure to be called. Note that if a virtual address is used, it must be expressed as an integer, not as an expression.

argument

One or more arguments required by the procedure.

Command Qualifiers

None.

Example

CALL SUB1(X)

```
DBG>
routine start at SUB1
value returned is 7FFF07DC
```

CANCEL

Description

Cancels breakpoints, tracepoints, and watchpoints, and restores scope and user-set entry/display modes and types to their default values. The item canceled depends on the keyword specified in the command.

See the individual command descriptions following for more information.

Format

CANCEL keyword [/qualifier] [parameters]

Command Parameters

keyword

Specifies the item to be canceled. Keyword can be ALL, BREAK, EXCEPTION BREAK, MODE, MODULE, SCOPE, SOURCE, TRACE, TYPE, or WATCH.

parameters

Depends on the keyword specified.

Command Qualifiers

Depends on the keyword specified.

CANCEL ALL

Description

Cancels all breakpoints, tracepoints, watchpoints, and restores scope and user-set entry/display modes and types to their default values.

CANCEL ALL does not affect the current language or the modules included in the debugger symbol table.

Format

CANCEL ALL

Command Parameters

None.

Command Qualifiers

None.

Example

```
DBG> CAN ALL
```

CANCEL BREAK

Description

Cancels a breakpoint at the location denoted by the specified address expression or cancels all breakpoints.

You must specify either an address expression or the /ALL qualfier, but not both, when you use the CANCEL BREAK command.

Note that the command CANCEL ALL also cancels all breakpoints.

Format

CANCEL BREAK[/qualifier] [address-expression]

Command Parameters

address-expression

Denotes the location of the breakpoint to be canceled.

Command Qualifiers

/ALL

Cancels all breakpoints.

Examples

1. DBG> CAN BRE MAIN\LOOP+10

2. DBG> CAN BRE/ALL

CANCEL EXCEPTION BREAK

Description

Cancels exception breakpoints.

The command CANCEL EXCEPTION BREAK cancels the effect of the command SET EXCEPTION BREAK. As a result of the command CANCEL EXCEPTION BREAK, exception conditions generated by your program are handled in the following way:

- The debugger fields and resignals the exception.
- If you have defined an exception handler in your program, it is executed.
- If you have not defined an exception handler or if an exception handler that you have defined resignals the exception, a diagnostic message is issued and control is returned to the debugger, which then displays its prompt.

Format

CANCEL EXCEPTION BREAK

Command Parameters

None.

Command Qualifiers

None.

Example

```
DBG> CAN EXC BRE
```

CANCEL MODE

Description

Cancels radix mode and symbolic/nonsymbolic mode settings established by the SET MODE command, thus reestablishing language-specific default mode values.

Format

CANCEL MODE

Command Parameters

None.

Command Qualifiers

None.

Example

```
DBG> CAN MO
```

CANCEL MODULE

Description

Removes symbols declared in the specified module or modules, or in all modules, from the Run-Time Symbol Table (RST).

The CANCEL MODULE command can be used when the symbol table is full and it is necessary to delete the symbols of one or more modules in order to make room for the symbols of one or more other modules.

You can remove the symbols from one module, from a list of modules, or from all modules.

Format

```
CANCEL MODULE[/qualifier] [ modname[,modname...] ]
```

Command Parameters

modname

Specifies the name of the module whose symbols are to be removed from the Run-Time Symbol Table.

Command Qualifiers

/ALL

Removes the symbols in all modules from the run-time symbol table.

Examples

1. DBG> CAN MODU SUB1

2. DBG> CAN MODU/ALL

CANCEL SCOPE

Description

Cancels the current scope search list established by the SET SCOPE command.

The CANCEL SCOPE command is equivalent in effect to the command SET SCOPE 0.

As a result of the CANCEL SCOPE command, symbols without pathname prefixes are interpreted as if they occurred in the most recent invocation of the currently active program unit, that is, the program unit containing the PC.

Format

CANCEL SCOPE

Command Parameters

None.

Command Qualifiers

None.

Example

```
DBG> CAN SCO
```

CANCEL SOURCE

Description

Cancels the current source directory search list established by previous SET SOURCE commands.

The CANCEL SOURCE command without a command qualifier cancels the effect of a previous SET SOURCE command; it does not cancel the effect of any previous SET SOURCE/MODULE=modname commands.

The CANCEL SOURCE/MODULE=modname command cancels the effect of a previous SET SOURCE/MODULE=modname command in which the same module name was specified; it does not cancel the effect of a previous SET SOURCE command or of a SET SOURCE/MODULE=modname command in which a different module name was specified.

When a SET SOURCE command is canceled, the debugger again expects each source file to be in the same directory as it was in at compile time.

Format

```
CANCEL SOURCE[/MODULE=modname]
```

Command Parameters

None.

Command Qualifiers

/MODULE=modname

Specifies the name of a module for which a source directory search list is to be canceled.

Example

```
DBG> SH SOU
source directory search list for COBOLTEST:
        SYSTEM::DEVICE:[PROJD]
        [014,015]
source directory search list for all other modules:
        [PROJA]
        [PROJB]
        [PETER.PROJC]
DBG> CAN SOU
DBG> SH SOU
source directory search list for COBOLTEST:
        SYSTEM::DEVICE:[PROJD]
        [014,015]
DBG> CAN SOU/MODU=COBOLTEST
DBG> SH SOU
no directory search list in effect
```

CANCEL TRACE

Description

Cancels a tracepoint at a location denoted by a specified address expression, at each member of the family of CALL instructions, at each member of the family of BRANCH instructions, or at all instructions.

If you specify an address expression as a parameter in the CANCEL TRACE command, the tracepoint at the location denoted by that address expression is canceled. If you specify /BRANCH or /CALL, tracepoints on instructions that are members of the family of BRANCH instructions or CALL instructions, respectively, are canceled. If you specify /ALL, all tracepoints are canceled.

Note that the command CANCEL ALL also cancels all tracepoints.

Format

```
CANCEL TRACE[/qualifier] [address-expression]
```

Command Parameters

address-expression

Denotes the location of the tracepoint to be canceled.

Command Qualifiers

/ALL

Cancels all tracepoints.

/BRANCH

Cancels tracepoints at members of the family of BRANCH instructions.

/CALL

Cancels tracepoints at members of the family of CALL instructions.

Examples

1. `DBG> CAN TRA MAIN\LOOP+10`

2. `DBG> CAN TRA/ALL`

CANCEL TYPE/OVERRIDE

Description

Cancels the debugger override type established by the SET TYPE/OVERRIDE command, thus setting the current override type to "none."

As a result of the CANCEL TYPE/OVERRIDE command, program entities are interpreted in compiler-generated types or in the default type.

Note that the CANCEL TYPE command must be specified with the /OVERRIDE command qualifier.

Format

```
CANCEL TYPE/OVERRIDE
```

Command Parameters

None.

Command Qualifiers

/OVERRIDE

Must be specified. The minimum abbreviation is /OVERR.

Example

```
DBG>CAN TYPE/OVERR
```

CANCEL WATCH

Description

Cancels a watchpoint set at the location denoted by the specified address expression or at all watchpoints.

If you specify an address expression as a parameter, the watchpoint at the location denoted by that address expression is canceled. If you specify /ALL, all watchpoints are canceled.

Note that the CANCEL ALL command also cancels all watchpoints.

Format

CANCEL WATCH[/qualifier] [address-expression]

Command Parameters

address-expression

Denotes the location of the watchpoint to be canceled.

Command Qualifiers

/ALL

Cancels all watchpoints.

Examples

1. DBG> CAN WAT SUB2\D

2. DBG> CAN WAT/ALL

CTRL/C, CTRL/Y, or CTRL/Z

Description

Unless the system or the user program has defined a CTRL/C service routine, the effect of pressing CTRL/Y or CTRL/C is the same: the image is interrupted but unchanged, the terminal type-ahead buffer is purged, and the command interpreter receives control.

Interrupting program execution with CTRL/Y or CTRL/C is useful if you have not run your program with the debugger but decide that you want the debugger, or if your program is executing an infinite loop that does not contain a breakpoint.

After interrupting your program with CTRL/Y or CTRL/C, you can start or restart the debugger by issuing the DCL command DEBUG.

Issuing CTRL/Z causes orderly termination of the debugging session. It is identical in effect to the EXIT command.

Format

CTRL/C
CTRL/Y
CTRL/Z

Command Parameters

None.

Command Qualifiers

None.

Examples

1.
```
DBG> CTRL/Y                  !Interrupt a debugging session
^Y                           !with CTRL/Y.
$
```

2.
```
$ DEBUG                      !Restart a debugging session with
DBG> CTRL/Z                  !the DCL command DEBUG; then end
$                            !the session with the CTRL/Z command.
```

DEFINE

Description

Defines one or more symbols and assigns them specified addresses for the duration of the debugging session.

You can use a defined symbol to refer to an address in the image. For example, you can define a symbol for a nonsymbolic program location or for a symbolic program location having a long pathname prefix; this enables you to refer to that program location by the newly defined symbol.

In symbol translation, the debugger searches symbols you define during the debugging session first. Consequently, if you define a symbol that already exists in your program, the debugger translates the symbol according to its defined definition, unless you specify a pathname prefix.

Once you have defined a symbol, you cannot explicitly cancel its definition but you can redefine it.

You can define more than one symbol in a single DEFINE command by separating symbols (and their equivalent expressions) with a comma (,).

You can use the EVALUATE command to determine the equivalence value of a symbol.

Format

```
DEFINE  symbol=expression[,symbol=expression...]
```

Command Parameters

symbol

Specifies the name of the symbol to be defined. The following rules must be followed when defining a symbol name:

- Symbols can be composed of alphanumeric characters (A-Z and 0-9), underscores (_), dollar signs ($), and periods (.). Periods cannot be used in some languages. The debugger interprets lowercase alphabetic characters as uppercase. Thus "LOOP" and "loop" are the same to the debugger.
- The first character must not be a number (0-9) or a period (.).
- The symbol must be no more than 31 characters long.

In addition, by DIGITAL convention:

- The dollar sign is reserved for names defined by DIGITAL. This convention ensures that a user-defined name will not conflict with a DIGITAL-defined name.
- The period (.) should not be used because most languages do not allow periods in symbol names.

expression

Either a virtual address or a symbolic address. The expression must have a value between 1 and 0FFFFFFFF (hexadecimal). Integers in the expression are interpreted in the current radix mode. You can use radix operators to override the current radix mode.

Command Qualifiers

None.

Examples

1. DBG>DEF CHK=MAIN\LOOP+10

2. DBG>DEF OLDR5=@R5, VALUE=15+30A

DEPOSIT

Description

Deposits the expression directly following the equal sign (=) into the location denoted by the address expression, and deposits any additional expressions specified into locations which are logical successors to the location denoted by the specified address expression.

If more than one expression is specified in the command, a comma (,) is used to separate them. Note however that some languages do not allow more than one expression.

Type command qualifiers may be used to associate a type with the location denoted by the address expression and with the location(s) denoted by logical successors (in the event that more than one expression is to be deposited).

Radix mode command qualifiers may be used to influence the interpretation of integers in both address expressions and expressions.

Format

```
DEPOSIT[/qualifier] address-expression = expression[,expression...]
```

Command Parameters

address-expression

Denotes the location into which the expression immediately following the equal sign (=) is to be deposited.

In some languages, the DEPOSIT command sets the current entity symbol (.) to the program location denoted by address expression. Logical successors are calculated using the value of the current entity symbol.

expression

Denotes the value to be deposited. If more than one expression is specified, the first expression is deposited at the location denoted by the address expression, the second expression at the location denoted by the logical successor, and so on, for each expression specified. Separate expressions with a comma (,).

Upon execution of the DEPOSIT command, the expression is evaluated in the syntax of the source language, and the value of the expression is converted to the type associated with the address expression and placed at the location denoted by the address expression.

Radix mode qualifiers usually affect the interpretation of an expression.

If the expression is an ASCII string or a VAX-11 instruction, it must be delimited by quotation marks (") or by apostrophes (').

In addition, if a particular language does not support a debugger type (such as one of the floating point, quadword integer, or octaword integer types), you may specify that type as a command qualifier in the DEPOSIT command only if you enclose the expression(s) to be deposited in quotation marks or apostrophes and only if the expression(s) contain no operators or delimiters. In this case, the debugger uses its own type-conversion rules rather than those of the language. For example,

```
DEPOSIT/FLOAT X = '123.5'
```

Command Qualifiers

/BYTE

Specifies that the type byte integer (length 1 byte) be associated with the location denoted by the address expression.

/WORD

Specifies that the type word integer (length 2 bytes) be associated with the location denoted by the address expression.

/LONG

Specifies that the type longword integer (length 4 bytes) be associated with the location denoted by the address expression.

/QUADWORD

Specifies that the type quadword integer (length 8 bytes) be associated with the location denoted by the address expression.

/OCTAWORD

Specifies that the type octaword integer (length 16 bytes) be associated with the location denoted by the address expression.

/FLOAT

Specifies that the F_floating type (length 4 bytes) be associated with the location denoted by the address expression. Values of type F_floating may range from .29*10**-38 to 1.7*10**38 with approximately 7 decimal digits precision.

/D_FLOAT

Specifies that the D_floating type (length 8 bytes) be associated with the location denoted by the address expression. Values of type D_floating may range from .29*10**-38 to 1.7*10**38 with approximately 16 decimal digits precision.

/G_FLOAT

Specifies that the G_floating type (length 8 bytes) be associated with the location denoted by the address expression. Values of type G_floating may range from .56*10**-308 to .9*10**308 with approximately 15 decimal digits precision.

/H_FLOAT

Specifies that the H_floating type (length 16 bytes) be associated with the location denoted by the address expression. Values of type H_floating may range from .84*10**-4932 to .59*10**4932 with approximately 33 decimal digits precision.

/ASCII:n

Specifies that the ASCII character type (length n bytes) be associated with the location denoted by the address expression. The value n is interpreted in the radix specified by a radix mode command qualifier or in the current radix mode. If n is omitted, the debugger assumes a default length of 4 bytes.

/INSTRUCTION

Specifies that the VAX-11 instruction type (variable length) be associated with the location denoted by the address expression. Length associated with type instruction is variable, depending on the number of operands and the kind of addressing modes used in the instruction.

/OCTAL

Specifies that numbers in expressions and in the address expression be interpreted in octal radix.

/DECIMAL

Specifies that numbers in expressions and in the address expression be interpreted in decimal radix.

/HEXADECIMAL

Specifies that numbers in expressions and in the address expression be interpreted in hexadecimal radix.

Examples

1. DBG> D X = A+2.5

2. DBG> D/BYTE WORK=212

3. DBG> D/OCTAWORD BIGINT='111222333444555'

4. DBG> D/FLOAT BIGFLT=1.11949*10**35

5. DBG> D/ASCII:10 WORK+20='abcdefghij'

6. DBG> D/INSTR SUB2+2 = 'MOVL #20A,R0', 'MOVL #3,R1'

EVALUATE

Description

Evaluates an expression. The debugger interprets the parameter in an EVALUATE command as a source-language expression, evaluates it in the syntax and semantics of the source language, and displays its value as a literal in the source language.

If an expression contains symbols with different compiler-generated types, the debugger uses the type-conversion rules of the current language to evaluate the expression.

If a radix mode command qualifier is specified, the debugger interprets literals in the expression in that radix and (in some languages) displays the value of the expression as a literal in that radix.

You can evaluate more than one expression in a single EVALUATE command by separating expressions with a comma (,). Note however that some languages do not allow more than one expression.

Using the EVALUATE command, you can perform arithmetic calculations that may or may not be related to your program. In effect, you can use the debugger as a calculator.

Format

```
EVALUATE[/qualifier] expression[,expression...]
```

Command Parameters

expression

Any legal expression in the source language, where an expression may be composed of numbers and one or more operators, operands, or delimiters.

Command Qualifiers

/DECIMAL

Specifies that numbers in the expression be interpreted in decimal radix and (in some languages) that the value of the expression be displayed in decimal radix.

/HEXADECIMAL

Specifies that numbers in the expression be interpreted in hexadecimal radix and (in some languages) that the value of the expression be displayed in hexadecimal radix.

/OCTAL

Specifies that numbers in the expression be interpreted in octal radix and (in some languages) that the value of the expression be displayed in octal radix.

Examples

1. DBG> EV 100.34 * (14.2 + 7.9)
   2217.514

2. DBG> EV/OCTAL X
   1512

EVALUATE/ADDRESS

Description

Evaluates an address expression and displays the results as a literal in the specified or default radix mode.

You can evaluate more than one address expression in a single EVALUATE/ADDRESS command by separating address expressions with a comma (,). Note however that some languages do not allow more than one address expression.

The EVALUATE/ADDRESS command is used to determine the virtual address(es) designated by the specified address expression(s).

If a radix mode command qualifier is specified, the debugger interprets literals in the address expression in the specified radix and displays the value of the expression as a literal in that radix.

Format

```
EVALUATE/ADDRESS address-expression[,address-expression...]
```

Command Parameters

address-expression

A language-independent address expression, which may be composed of numbers and/or symbols, as well as one or more operators, operands, or delimiters.

Command Qualifiers

/DECIMAL

Specifies that numbers in the address expression(s) be interpreted in decimal radix and that the value of the address expression(s) be displayed in decimal radix.

/HEXADECIMAL

Specifies that numbers in the address expression(s) be interpreted in hexadecimal radix and that the value of the address expression(s) be displayed in hexadecimal radix.

/OCTAL

Specifies that numbers in the address expression(s) be interpreted in octal radix and that the value of the address expression(s) be displayed in octal radix.

Examples

1. DBG> EV/A MODNAME\%LINE 110

2. DBG> EV/A/HEX Y

3. DBG> EV/A A,B,C

EXAMINE

Description

Displays the value of the entity at the location denoted by the address expression in the type associated with that location.

You can examine more than one entity in a single EXAMINE command by entering more than one address expression, separating each with a comma (,).

You can examine a range of entities in a single EXAMINE command by entering the address expression that denotes the first entity in the range, a colon (:), and the address expression that denotes the last entity in the range. Note however that some languages do not allow the examination of ranges.

You can examine a list of ranges of entities in a single EXAMINE command by separating each range with a comma.

If you specify a type command qualifier in the EXAMINE command, the debugger displays the entity or entities in that type.

If you specify a radix mode command qualifier in the EXAMINE command, the debugger interprets numeric literals in the address expression(s) in the specified radix and displays the examined entity or entities in the specified radix.

If you specify the /SYMBOL command qualifier in the EXAMINE command, the debugger displays address expression(s) and instruction operand(s) symbolically (if possible). If you specify the /NOSYMBOL command qualifier, the debugger displays symbolic information in its numeric equivalent.

When you examine the PSL in SYMBOL mode, the debugger displays its contents in a formatted arrangement; otherwise, the debugger displays its contents as a numeric literal in the current radix mode.

If you specify the /SOURCE command qualifier, the debugger displays the source line(s) corresponding to the location(s) designated by the address expression(s).

Format

```
EXAMINE[/qualifier[/qualifier...]] -
        [address-expression[:address-expression]-
        [,address-expression[:address-expression]...]]
```

Command Parameters

address-expression

Specifies the entity to be examined.

The debugger evaluates an address expression according to its own language-independent rules to yield a virtual address and interprets that address as the location at which the entity to be examined resides.

An address expression may be composed of numbers and/or symbols, as well as one or more operators, operands, or delimiters.

If a range of entities is to be examined, the value of the address expression that denotes the first entity in the range must be less than the value of the address expression that denotes the last entity in the range. If a list of entities is to be examined, the address expressions that denote them may be listed in the command in any order.

Command Qualifiers

/BYTE

Specifies that the examined entity or entities be displayed in the type byte integer (length 1 byte).

/WORD

Specifies that the examined entity or entities be displayed in the type word integer (length 2 bytes).

/LONG

Specifies that the examined entity or entities be displayed in the type long integer (length 4 bytes).

/QUADWORD

Specifies that the examined entity or entities be displayed in the type quadword integer (length 8 bytes).

/OCTAWORD

Specifies that the examined entity or entities be displayed in the type octaword integer (length 16 bytes).

/FLOAT

Specifies that the examined entity or entities be displayed in the F_floating type (length 4 bytes). Values of type F_floating may range from .29*10**-38 to 1.7*10**38 with approximately 7 decimal digits precision.

/D_FLOAT

Specifies that the examined entity or entities be displayed in the D_floating type (length 8 bytes). Values of type D_floating may range from .29*10**-38 to 1.7*10**38 with approximately 16 decimal digits precision.

/G_FLOAT

Specifies that the examined entity or entities be displayed in the G_floating type (length 8 bytes). Values of type G_floating may range from .56*10**-308 to .9*10**308 with approximately 15 decimal digits precision.

/H_FLOAT

Specifies that the examined entity or entities be displayed in the H_floating type (length 16 bytes). Values of type H_floating may range from .84*10**-4932 to .59*10**4932 with approximately 33 decimal digits precision.

/ASCII:n

Specifies that the examined entity or entities be displayed in the ASCII character type (length n bytes). The length indicates both the number of bytes of memory to be examined and the number of ASCII characters to be displayed. If n is omitted, the debugger uses the default value of 4 bytes.

/INSTRUCTION

Specifies that the examined entity or entities be displayed in the type VAX-11 instruction (variable length). The length associated with the instruction type is variable, depending on the number of operands and the kind of addressing modes used in the instruction.

/OCTAL

Specifies that numeric literals in the address expression(s) be interpreted in octal radix and that the examined entity or entities be displayed in octal radix.

/DECIMAL

Specifies that numeric literals in the address expression(s) be interpreted in decimal radix and that the examined entity or entities be displayed in decimal radix.

/HEXADECIMAL

Specifies that numeric literals in the address expression(s) be interpreted in hexadecimal radix and that the examined entity or entities be displayed in hexadecimal radix.

/SYMBOL

Specifies that the debugger translate the value of the address expression in the EXAMINE command and use its symbolic equivalent (if one exists) on the next display line when it displays the entity at the location denoted by the address expression. If the displayed entity is of type instruction, its operand(s) are displayed symbolically (if possible). The PSL is displayed in formatted form when the /SYMBOL command qualifier is specified in an EXAMINE command.

/NOSYMBOL

Specifies that the debugger display the value of the address expression in the EXAMINE command as a virtual memory address or as an offset from a symbolic location when it displays the entity at that location on the display line. If the displayed entity is of type instruction, its operand(s) are displayed as numeric literals, even if they have a symbolic representation. The PSL is displayed as a numeric literal, not in formatted form.

/SOURCE

Specifies that the debugger display the source line(s) corresponding to the location(s) designated by the address expression(s).

Examples

```
1. DBG>EX/ASC WORK+20
   DETAT\WORK+20:  abcd

2. DBG>E/WORD MAIN
   MAIN\MAIN:  483C

3. DBG>E/I MAIN+2
   MAIN\MAIN+02:  MOVAL   L^MAIN\A,R11

4. DBG>E/DEC/WORD WORK:WORK+6 , SUB1\D
   DETAT\WORKDATA: 256
   DETAT\WORKDATA+2:  770
   DETAT\WORKDATA+4:  1284
   DETAT\WORKDATA+6:  1798
   SUB1\D:  31

5. DBG>E/NOSYM WORKDATA
   0000086F:  03020100
```

EXIT

Description

Ends the debugging session, or ends the execution of commands in a command procedure or DO command sequence.

When you issue the EXIT command at the terminal, you cause orderly termination of the debugging session: the debugger exit handler is executed, exit status information is displayed, and control is returned to the DCL command interpreter. You cannot continue to debug your program by issuing the DCL commands DEBUG or CONTINUE. To restart the debugger, you must run the program again.

When the debugger executes an EXIT command in a command procedure, control returns to the command stream that invoked the command procedure. For example, if the command procedure was invoked from within a DO command sequence, control returns to that DO command sequence, where the debugger executes the next command (if any remain in the sequence).

When the debugger executes an EXIT command in a DO command sequence, it ignores any remaining commands in that sequence and displays its prompt.

Format

EXIT

Command Parameters

None.

Command Qualifiers

None.

Example

```
DBG> EXIT
$
```

GO

Description

Starts or continues program execution.

If the GO command is specified without an address expression as a parameter, execution resumes at the point of suspension or, in the case of debugger start up, at the transfer address.

If the GO command is specified with an address expression as a parameter, execution resumes at the location denoted by the address expression. Note that using an address expression as a parameter in the GO command can produce unpredictable results if the address expression denotes a program location that has not yet executed.

Format

```
GO [address-expression]
```

Command Parameters

address-expression

Specifies that program execution resume at the location denoted by the address expression.

Command Qualifiers

None.

Examples

1.
```
DBG> GO
routine start at MAIN\MAIN
routine break at SUB1\SUB1
```

2.
```
DBG> GO MAIN
routine start at MAIN\MAIN
routine break at SUB1\SUB1
```

3.
```
DBG> GO
routine start at SUB1\SUB1
%DEBUG-I-EXITSTATUS, is '%SYSTEM-S-NORMAL, normal successful
completion'
```

HELP

Description

Displays the following information about any debugger command: a description of the command, format of the command, parameters that may be specified with the command, and qualifiers that may be specified with the command.

If you want information about a particular qualifier or parameter, specify it as a subtopic. If you want information about all command qualifiers, specify "qualifier" as a subtopic. If you want information about all parameters, specify "parameter" as a subtopic. If you want information about all parameters and all qualifiers, specify an asterisk (*) as a subtopic.

Format

HELP topic [subtopic...]

Command Parameters

topic

Specifies the name of the command with which you need help.

subtopic

Specifies a particular qualifier or parameter about which you want further information, or a command keyword that gives you information about a range of qualifiers or parameters or both.

If you want information about a particular qualifier or parameter, specify it as a subtopic. Note that when you specify a qualifier, you must include the preceding slash (/). If you want information about all command qualifiers, specify "qualifier" as a subtopic. If you want information about all parameters, specify "parameter" as a subtopic. If you want information about all parameters and qualifiers, specify an asterisk (*) as a subtopic.

Command Qualifiers

None.

Example

```
DBG>HELP DEFINE

   DEFINE

        Defines one or more symbols and  assigns  them  specified
        addresses for the duration of the debugging session.

   Format:

        DEFINE symbol=expression [,symbol=expression...]

   Additional information available:

   Parameters
```

SEARCH

Description

Directs the debugger to search the source code for the specified string and to display the source line or lines containing an occurrence of the string.

SEARCH command qualifiers determine whether the debugger searches for all occurrences (/ALL) of the string or only the next occurrence (/NEXT) of the string, and whether the debugger displays any occurrence of the string (/STRING) or only those occurrences in which the string is not bounded on either side by a character that can be part of an identifier in the current language (/IDENTIFIER).

The range parameter designates a program region to be searched. You can search any module in the program, from the beginning or at a specified line number to the end or to a specified line number.

The string parameter specifies the source code characters for which to search. If the string parameter is not specified, the debugger uses the last specified search string, that is, the string parameter specified in the last SEARCH command. If there was no previous search string, an error message results.

Format

```
SEARCH[/qualifier[/qualifier]] range string
```

Command Parameters

range

Limits the debugger's search for occurrences of the string to specified program regions. These program regions may be specified in any of the following formats:

- MODNAME indicates a search of the specified module from line number 0 to the end of the module.
- MODNAME\LINE-NUM indicates a search of the specified module from the specified line number to the end of the module.
- MODNAME\LINE-NUM:LINE-NUM indicates a search of the specified module beginning at the line number specified to the left of the colon and ending at the line number specified to the right of the colon.
- LINE-NUM indicates a search of the module designated by the current scope setting from the specified line number to the end of the module.
- LINE-NUM:LINE-NUM indicates a search of the module designated by the current scope setting beginning at the line number specified to the left of the colon and ending at the line number to the right of the colon.

- NULL (that is, no entry) indicates a search of the same module as that from which a source line was most recently displayed (as a result of a TYPE, EXAMINE/SOURCE, or SEARCH command, for example), beginning at the first line following the line most recently displayed and continuing to the end of the module.

string

Specifies the string in the source code for which to search.

The string parameter may be enclosed in quotation marks (") or in apostrophes (') or may be specified without delimiters.

If the string is delimited by either quotation marks or apostrophes, it may contain spaces and tabs, as well as any alphanumeric or special characters.

If the string is enclosed in quotation marks, an enclosed quotation mark is indicated by a double quotation mark (""). If the string is enclosed in apostrophes, an enclosed apostrophe is indicated by a double apostrophe ('').

If, however, the string is not delimited, the following restrictions apply:

- It must have no leading or trailing blanks or tabs.
- It must have no embedded semicolon (;).
- The range parameter must not be null; that is, an explicit range must be specified.

Command Qualifiers

/ALL

Specifies that the debugger search for all occurrences of the string in the specified range and display every line containing an occurrence of the string.

/NEXT

Specifies that the debugger search for the first occurrence of the string in the specified range and only display the line containing this occurrence. This is the default.

/IDENTIFIER

Specifies that the debugger search for an occurrence of the string in the specified range but display the string only if it is not bounded on either side by a character that can be part of an identifier in the current language.

/STRING

Specifies that the debugger search for and display the string as specified, and not interpret the context surrounding an occurrence of the string, as it does in the case of /IDENTIFIER. This is the default.

Examples

```
1.  DBG> SEA/S/A 40:50 D
    module COBOLTEST
         40: 02       D2N      COMP-2 VALUE -234560000000.
         41: 02       D        COMP-2 VALUE  222222.33.
         42: 02       DN       COMP-2 VALUE -222222.333333.
         47: 02       DR0      COMP-2 VALUE  0.1.
         48: 02       DR5      COMP-2 VALUE  0.000001.
         49: 02       DR10     COMP-2 VALUE  0.00000000001.
         50: 02       DR15     COMP-2 VALUE  0.0000000000000001.

2.  DBG> SEA/I/A 40:50 D
    module COBOLTEST
         41: 02       D        COMP-2 VALUE  222222.33.

3.  DBG> SEA/N 40:50 D
    module COBOLTEST
         40: 02       D2N      COMP-2 VALUE -234560000000.
    DBG> SEA/N
    module COBOLTEST
         41: 02       D        COMP-2 VALUE  222222.33.

4.  DBG> SEA 80:90
    module COBOLTEST
         80: 02       LS10     PIC S9(10)      LEADING SEPARATE VALUE
           -: 1234567890.

5.  DBG> SEA COBOLTEST\170 'D'
    module CCBOLTEST
       170:           DISPLAY ES1.

6.  DBG> SEA COBOLTEST\150 ' E'
    module COBOLTEST
       161:           MOVE SC1 TO ES0.

7.  DBG> SEA/N 1 ". "
    module COBOLTEST
          2: PROGRAM-ID. COBOLTEST.
```

SET

Description

Establishes breakpoints, tracepoints, or watchpoints, or sets the language, modules, step conditions, source display, modes, and types. The item set depends on the keyword specified.

See the individual command descriptions following for more information.

Format

SET keyword[/qualifier] parameter

Command Parameters

keyword

Specifies the item to be set. Keyword can be BREAK, EXCEPTION BREAK, LANGUAGE, LOG, MARGIN, MAX_SOURCE_FILES, MODE, MODULE, OUTPUT, SCOPE, SEARCH, SOURCE, STEP, TRACE, TYPE, or WATCH.

parameters

Depends on the keyword specified.

Command Qualifiers

Depends on the keyword specified.

SET BREAK

Description

Establishes a breakpoint at the location denoted by the specified address expression.

When a breakpoint is activated, the debugger takes the following action:

1. Suspends program execution
2. Displays the location of the breakpoint (and displays the line of source code corresponding to that instruction if the SOURCE parameter is in effect by virtue of a previous SET STEP SOURCE command)
3. Executes a DO command sequence (if one was specified when the breakpoint was set)
4. Issues its prompt

When you set a breakpoint, you can specify that one or more debugger commands be executed at breakpoint activation. You do this by including a DO command sequence in the SET BREAK command.

By specifying the /AFTER:n command qualifier in the SET BREAK command, you can direct the debugger to ignore the first n-1 activations of the breakpoint location and take break action only on the nth and subsequent activations of the specified location.

Format

```
SET BREAK[/qualifier] address-expression [DO(command[;command...])]
```

Command Parameters

address-expression

Specifies the location at which the breakpoint is to be set.

command

Any debugger command that you want the debugger to execute as part of the DO command sequence when break action is taken. If you specify more than one command, separate them with a semicolon (;).

Command Qualifiers

/AFTER:n

Specifies that break action not be taken until the nth and subsequent activations of the location designated by the address expression. The value n is interpreted in decimal radix mode, no matter what radix mode settings are in effect.

Examples

1. DBG> SE BREAK/AFTER:3 SUB2

2. DBG> SE BR LOOP1 DO (EX D; STEP; EX Y; GO)

3. DBG> SE B 1440

SET EXCEPTION BREAK

Description

Specifies that the debugger treat an exception condition generated by your program as a breakpoint.

As a result of this command, whenever your program generates an exception condition, the debugger responds by suspending program execution, reporting the exception condition, and prompting you for input.

Whenever an exception breakpoint is activated, therefore, you have the opportunity to issue debugger commands. When you want to continue program execution, you can issue one of the following commands:

- A GO command without an address-expression parameter. In this case, the debugger fields and resignals the exception, thus allowing any user-declared exception handlers to execute.

- A GO command with an address-expression parameter. In this case, the debugger allows program execution to continue at the specified location, thus inhibiting the execution of any user-declared exception handlers.

Note that you cannot issue a STEP command to continue program execution after an exception breakpoint is activated. The debugger issues a warning message because the STEP command is illegal in this context.

Format

SET EXCEPTION BREAK

Command Parameters

None.

Command Qualifiers

None.

Example

```
DBG> SET EXC BRE
```

SET LANGUAGE

Description

Establishes the current language.

At debugger start up, the debugger sets the current language to that in which the module containing the transfer address is written. You can (and should) change this language by the SET LANGUAGE command if you begin debugging a module written in a different source language.

The current language influences several debugger .default parameters, namely, those of type, radix, and step values. In addition, languages differ on such matters as type conversions; the evaluation of expressions; and acceptable syntax, special characters, and symbols.

Format

SET LANGUAGE language-name

Command Parameters

language-name

Specifies the name of the language. Valid languages include the following: BASIC, BLISS, COBOL, FORTRAN, PASCAL, PLI, and MACRO.

Command Qualifiers

None.

Examples

1. DBG>SET LANG COBOL

2. DBG>SET LANG PASCAL

SET LOG

Description

Specifies the name of the log file to which the debugger writes when the output parameter is set to LOG.

If the output parameter is set to LOG but no log file is specified by the SET LOG command, the debugger writes to the file DEBUG.LOG by default.

If the debugger is writing to a log file and you specify another log file by the SET LOG command, the debugger closes the former file and begins writing to the file specified in the SET LOG command.

If you specify a file name but no file type in the SET LOG command, the debugger assumes a default file type of LOG.

If you specify a version number in the file specification parameter in the SET LOG command and that version of the file already exists, the debugger cannot open a new file. Instead, the debugger writes to the file specified, appending the log of the debugging session onto the end of that file.

Note that the SET LOG command only determines the name of a log file; it does not cause the debugger to create or write to the specified file. The SET OUTPUT LOG command accomplishes that.

Format

SET LOG file-spec

Command Parameters

file-spec

The file specification of the log file. If the file specification begins with a special symbol such as a square bracket ([), you must enclose the file specification in quotation marks (") or apostrophes ('); otherwise, do not enclose the file specification in these delimiters.

Command Qualifiers

None.

Examples

1. DBG>SET LOG CALC

2. DBG>SET LOG "[CODEPROJ]FEB29.TMP"

SET MARGIN

Description

Specifies the leftmost source-line character position at which to begin display of a line of source code (the left margin) and/or the rightmost character position at which to end display of a line of source code.

By default, the debugger displays a source line beginning at character position 1 of the source line. Source-line character position 1 is actually character position 9 on your terminal screen. The first 8 character positions on the terminal screen are occupied by the line number and cannot be manipulated by the SET MARGIN command.

Increasing the left margin setting is particularly useful when the source code is deeply indented; by eliminating the display of empty space, more space is available on the terminal line for the display of source code.

Decreasing the right margin setting (from its default value of 255) prevents wrapping of long lines by truncating them. Since a wrapped line of source code requires two lines on the terminal screen, eliminating wrapping allows more lines of source code to be displayed on the terminal screen.

The SET MARGIN command affects only the display of source lines, that is, the display resulting from commands such as TYPE and EXAMINE/SOURCE. The SET MARGIN command does not affect the display resulting from commands (such as EXAMINE, EVALUATE, SHOW MODE, and so on) that do not display source code. If a command displays source code together with other information -- as, for example, STEP/SOURCE does -- the display of source code is affected by the current margin settings but the other information is not.

Format

```
                   ( rm      )
SET MARGIN         < lm:rm   >
                   ( lm:     )
                   (    :rm  )
```

If you specify a single number, the debugger sets the left margin to 1 and the right margin to the number specified.

If you specify two numbers, separated with a colon (:), the debugger sets the left margin to the number specified to the left of the colon and the right margin to the number specified to the right of the colon.

If you specify a single number followed by a colon, the debugger sets the left margin to the number specified and leaves the right margin unchanged.

If you specify a colon followed by a single number, the debugger sets the right margin to the number specified and leaves the left margin unchanged.

Command Parameters

lm

The source-line character position at which to begin display of the line of source code (the left margin).

rm

The source-line character position at which to end display of the line of source code (the right margin).

Command Qualifiers

None.

Examples

```
1.  DBG> SH MAR
    left margin: 1 , right margin: 255
    DBG> TYPE 116
    module COBOLTEST
       116: 02      RA26    PIC A(26)       JUST RIGHT VALUE
         -: "ABCDEFGHIJKLMNOPQRSTUVWXYZ".

2.  DBG> SET MARGIN 50
    DBG> SH MAR
    left margin: 1 , right margin: 50
    DBG> TY 116
    module COBOLTEST
       116: 02      RA26    PIC A(26)       JUST RIGHT VALUE

3.  DBG> SET MARGIN 10:60
    DBG> SH MAR
    left margin: 10 , right margin: 60
    DBG> TY 116
    module COBOLTEST
       116: A26    PIC A(26)       JUST RIGHT VALUE   "ABCDEFGHI

4.  DBG> SET MARGIN :100
    DBG> SH MAR
    left margin: 10 , right margin: 100

5.  DBG> SET MARGIN 5:
    DBG> SH MAR
    left margin: 5 , right margin: 100

6.  DBG> SE MAR 124
    DBG> SH MAR
    left margin: 1, right margin: 124
```

SET MAX_SOURCE_FILES

Description

Specifies the maximum number of source files that the debugger may keep open at any one time.

By default, the debugger may keep five source files open at any one time.

Opening a source file requires the use of an I/O channel, which is a limited system resource. Both the program and the debugger use I/O channels. To ensure that the debugger does not use all available I/O channels and thus cause the program to fail (for lack of an available I/O channel), you can issue the SET MAX_SOURCE_FILES command to specify the maximum number of source files (and thus source file I/O channels) that the debugger may use at any one time.

Note that the value of MAX_SOURCE_FILES does not limit the number of source files that the debugger can open; rather, it limits the number that may be kept open at any one time. Thus, if the debugger reaches this limit, it must close a file in order to open another one.

Note too that setting MAX_SOURCE_FILES to a very small number can make the debugger's use of source files inefficient.

Format

```
SET MAX_SOURCE_FILES n
```

Command Parameters

n

Specifies the maximum number of source files that the debugger may keep open at any one time. The value of n may not exceed 20. The default value is 5.

Command Qualifiers

None.

Example

```
DBG> SHOW MAX SOURCE FILES
max_source_files: 5
DBG> SET MAX SOURCE FILES 8
DBG> SHOW MAX SOURCE FILES
max_source_files: 8
```

SET MODE

Description

Establishes the default radix and/or the default symbolic/nonsymbolic modes.

When you specify a radix mode in the SET MODE command, the debugger interprets numeric literals as values in that radix (unless you specify another radix mode as a command qualifier) and, in some languages, displays numeric values in that radix (unless you specify another radix mode as a command qualifier).

When the default mode is symbolic, the debugger displays the locations denoted by address expressions symbolically (if possible), displays instruction operands symbolically (if possible), and displays the value of the PSL in a formatted arrangement.

When the default mode is nonsymbolic, the debugger displays all symbols in their numeric equivalents.

Note that unlike the radix mode parameters, symbolic and nonsymbolic modes do not affect the interpretation of information you enter; they only affect the debugger's display of information.

You can specify more than one mode in a single SET MODE command by separating them with a comma.

Default modes set by the SET MODE command can be overriden by mode command qualifiers.

Format

```
SET MODE mode-keyword[,mode-keyword]
```

Command Parameters

mode-keyword

May be any one of the following:

- DECIMAL -- Sets the default radix to decimal
- HEXADECIMAL -- Sets the default radix to hexadecimal
- OCTAL -- Sets the default radix to octal
- NOSYMBOL -- Specifies that the debugger display all symbols in their numeric equivalents
- SYMBOL -- Specifies that the debugger display the locations denoted by address expressions symbolically (if possible), display instruction operands symbolically (if possible), and display the value of the PSL in a formatted arrangement

Command Qualifiers

None.

Example

```
DBG>SET MODE DEC, NOSYM
```

SET MODULE

Description

Copies into the Run-Time Symbol Table (RST) information about symbols in the specified module(s) or in all modules.

Symbol records must be present in the RST in order for the debugger to interpret those symbols in a debugging session. At debugger start up, symbol records for the module containing the transfer address are copied into the RST. To include symbol records for other modules, specify those modules in the SET MODULE command.

The RST is large enough to accommodate symbol records for the six largest modules in your program providing that your process quota is not exceeded.

Note that if a parameter in the SET SCOPE command designates a program location in a module whose symbol records are not already in the RST, the debugger copies the symbol records for that module into the RST when the SET SCOPE command is executed.

Format

```
SET MODULE[/qualifier] [modname[,modname...]]
```

Command Parameters

modname

Specifies the name of the module whose symbol records are to be copied into the RST.

Command Qualifiers

/ALL

Specifies that the symbol records of all modules be copied into the RST.

Examples

1. DBG> SET MODU SUB1

2. DBG> SET MODU/ALL

SET OUTPUT

Description

Specifies the debugger output configuration, that is, the way in which debugger responses to commands are displayed and recorded.

You can direct the debugger to write its output to a terminal, to a log file, or to both. Additionally, you can direct the debugger to include, in its output, each input command string in any command procedure or DO command sequence it is executing.

By default, the debugger writes output to the terminal, not to a log file, and does not include in its output each input command string in any command procedures or DO command sequences.

To change the default output configuration, use the SET OUTPUT command with any of the six parameters: TERMINAL, NOTERMINAL, LOG, NOLOG, VERIFY, NOVERIFY.

You may specify more than one parameter in a single SET OUTPUT command by separating parameters with a comma.

Format

```
SET OUTPUT parameter[,parameter...]
```

Command Parameters

parameter

May be one of the following six output parameters:

- LOG -- Specifies that both debugger output and user input be recorded in a log file. If you specify the name of the log file by the SET LOG command, the debugger writes to that file; otherwise, the debugger writes to a file with the default file specification DEBUG.LOG.
- NOLOG -- Specifies that debugger output and user input not be recorded in a log file. This is a default output parameter.
- TERMINAL -- Specifies that debugger output be displayed at the terminal. This is a default output parameter.
- NOTERMINAL -- Specifies that debugger output, except for diagnostic messages, not be displayed at the terminal.
- VERIFY -- Specifies that the debugger include, in its output, each input command string in any command procedure or DO command sequence it is executing.
- NOVERIFY -- Specifies that the debugger not include, in its output, each input command string in any command procedure or DO command sequence it is executing. This is a default output parameter.

Command Qualifiers

None.

Example

```
DBG> SET OUT VER,LOG,NOTERM
```

SET SCOPE

Description

Designates one or more program locations (specified by pathnames and/or other special characters) to be used in the interpretation of symbols that are specified without pathname prefixes in debugger commands.

The debugger attempts to interpret symbols without pathname prefixes by interpreting the symbol as if it appeared in the location denoted by the first parameter in the SET SCOPE command. If the debugger cannot interpret the symbol using the first parameter, it uses the next parameter in the SET SCOPE command, and continues using parameters in order of their specification, until it successfully interprets the symbol or until it exhausts the parameters specified.

When you specify more than one parameter in the SET SCOPE command, you establish a scope search list for the interpretation of symbols without pathname prefixes.

In addition to pathname prefixes, the backslash (\) and the set of decimal integers (0,1,2,...) may also be used as parameters in the SET SCOPE command. The backslash (\) represents the set of all program locations in which the declaration of any global symbol in the program is known. The decimal integers (0,1,2,...) represent invocations of the currently active program unit; zero (0) represents the most recent invocation, one (1) represents the next most recent invocation, and so on.

Note that when you want to set scope to a routine, you should specify both the routine name and the name of the module that contains that routine. Doing so is good practice even when the routine and module names are different; it is essential when the routine and module names are the same. For example, to set the scope to routine MAIN in module MAIN, you would issue the command SET SCOPE MAIN\MAIN.

Format

```
SET SCOPE location[,location...]
```

Command Parameters

location

Designates one or more program locations specified by pathnames and/or other special characters. The following are acceptable parameters in the SET SCOPE command:

- Pathname -- One or more program location labels, separated by the backslash character (\), that identifies a program location or a range of program locations. Program location labels may be module, routine, and block names, as well as line numbers and numeric labels. A common pathname format is the following: MODULE\ROUTINE\BLOCK\.

- 0,1,2,...-- The set of decimal integers represent invocations of the currently active program unit; zero (0) represents the most recent invocation, one (1) represents the next most recent invocation, and so on.

- \ -- Represents the set of all program locations in which the declaration of any global symbol in the program is known.

Command Qualifiers

None.

Example

```
DBG> SET SCOPE MOD\%LINE 15, 0
```

SET SEARCH

Description

Establishes current SEARCH parameters to be used by the debugger whenever a SEARCH command qualifier is not specified in a SEARCH command.

SEARCH parameters determine whether the debugger searches for all occurrences (ALL) of the string or only the next occurrence (NEXT) of the string, and whether the debugger displays any occurrence of the string (STRING) or only those occurrences in which the string is not bounded on either side by a character that can be part of an identifier in the current language (IDENTIFIER).

If you do not specify SEARCH parameters with the SET SEARCH command, the debugger uses the default values NEXT and STRING.

You can override current SEARCH parameters for the duration of a single SEARCH command by specifying other SEARCH parameters as command qualifiers in the SEARCH command.

You can specify more than one SEARCH parameter in a single SET SEARCH command by separating each parameter with a comma.

Format

SET SEARCH parameter[,parameter]

Command Parameters

parameter

Specifies the current SEARCH parameters. The following parameters may be specified:

- ALL specifies that the debugger search for all occurrences of the string in the specified range and display every line containing an occurrence of the string.
- NEXT specifies that the debugger search for the first occurrence of the string in the specified range and display the line containing this occurrence. This is the default.
- IDENTIFIER specifies that the debugger search for an occurrence of the string in the specified range but display the string only if it is not bounded on either side by a character that can be part of an identifier in the current language.
- STRING specifies that the debugger search for and display the string as specified, and not interpret the context surrounding an occurrence of the string, as it does in the case of IDENTIFIER.

Command Qualifiers

None.

Example

```
DBG> SHOW SEARCH
search settings: search for next occurence, as a string
DBG> SET SEARCH IDENT
DBG> SHOW SEARCH
search settings: search for next occurence, as an identifier
DBG> SET SE AL
DBG> SH SEA
search settings: search for all occurences, as an identifier
```

SET SOURCE

Description

Directs the debugger to search the specified directory or directories for source files. SET SOURCE is used to direct the debugger in the location of a source file that has been moved to another directory since being compiled.

By default, the debugger expects a source file to be in the same directory it was in at compile time.

If you specify more than one directory in a single SET SOURCE command, separating each directory name with a comma, you create a source directory search list. The debugger handles a source directory search list by searching the first directory specified to locate the source file for a module, then the second directory specified, then the next, and so on, until it either locates the source file or exhausts the list of directories.

If you specify the /MODULE=modname command qualifier in the SET SOURCE command, the debugger uses the specified directory search list only when locating source files for the specified module.

If you issue the SET SOURCE command without the /MODULE=modname command qualifier, the debugger uses the specified directory search list to locate source files for all modules that were not mentioned in a previous SET SOURCE/MODULE=modname command.

Format

```
SET SOURCE[/MODULE=modname] dirname[,dirname...]
```

Command Parameters

dirname

Specifies the directory to be searched.

Note that dirname may consist of one, several, or all fields in the full file specification, though typically it is only a directory name. A full file specification has the following format:

```
node::device:[directory]file-name.file-type;version-number
```

When specifying any of these fields, you must include the punctuation for that field, as shown in the above format.

Command Qualifiers

/MODULE=modname

Specifies that the indicated directory search list is to be used in locating source files only for the specified module.

Examples

1.

```
DBG> SHOW SOURCE
no directory search list in effect
DBG> SET SOURCE [PROJA],[PROJB],DISK:[PETER.PROJC]
DBG> SH SOU
source directory search list for all modules:
        [PROJA]
        [PROJB]
        DISK:[PETER.PROJC]
```

2.

```
DBG> SE SOU/MODU=COBOLTEST DEVICE:[PROJD],[014,015]
DBG> SH SOU
source directory search list for COBOLTEST:
        DEVICE:[PROJD]
        [014,015]
source directory search list for all other modules:
        [PROJA]
        [PROJB]
        DISK:[PETER.PROJC]
```

SET STEP

Description

Establishes current STEP parameters to be used by the debugger whenever a step command qualifier is not specified in a STEP command.

The parameters specified in the SET STEP command determine whether the debugger steps by line or by instruction, whether the debugger steps "into" or "over" called routines in the user program space, whether the debugger steps "into" or "over" called routines in system space, and whether the debugger displays lines of source code as it steps.

Each language establishes default STEP parameters. When you change the current language, STEP parameters are set to those specified in that language. You can change these STEP parameters by issuing the SET STEP command.

You can specify more than one parameter in a single SET STEP command by separating parameters with a comma.

Format

```
SET STEP parameter[,parameter...]
```

Command Parameters

parameter

Specifies a parameter that influences the debugger's behavior when a STEP command is issued. The following parameters may be specified in the SET STEP command.

- INSTRUCTION -- A single step causes execution of a single instruction at the location where execution was last suspended.
- LINE -- A single step causes execution of all the user code between the location where execution was last suspended and the next line boundary.
- INTO -- Steps "into" called routines in the user program space (and "into" called routines in system space if the SYSTEM parameter is also in effect). That is, in its execution of steps, the debugger does not differentiate between code within a called routine and code outside of a called routine.

- OVER -- Steps "over" called routines both in the user program space and in system space. That is, the debugger considers any code executed as the result of a CALL instruction, up to and including the corresponding RETURN instruction, to be part of a single STEP. Note that the routine itself is still executed.

- SYSTEM -- Steps "into" called routines in system space, provided that the INTO parameter is also in effect. That is, in its execution of steps, the debugger does not differentiate between code within a called routine in system space and code outside a called routine in system space. Thus, execution of a STEP command may result in suspension of execution in system space.

- NOSYSTEM -- Steps "over" called routines in system space. That is, the debugger considers any system-space code executed as the result of a CALL instruction, up to and including the corresponding RETURN instruction, to be part of a single STEP. Note that the routine itself is still executed.

- SOURCE -- Displays the line of source code that corresponds to the instruction(s) being executed with each step. Note that the SOURCE parameter also causes the debugger to display lines of source code when a breakpoint or watchpoint is activated.

- NOSOURCE -- Specifies that lines of source code not be displayed when STEP commands are executed. Note that the NOSOURCE parameter also causes the debugger not to display lines of source code when a breakpoint or watchpoint is activated. This is the default.

Command Qualifiers

None.

Examples

1. DBG> SET STEP INS,INTO

2. DBG> SET STEP SOURCE

SET TRACE

Description

Establishes a tracepoint at the location denoted by a specified address expression, or at instructions that are members of the family of CALL instructions, or at instructions that are members of the family of BRANCH instructions, or at instructions that are members of either the CALL or BRANCH families.

When a tracepoint is activated, the debugger suspends execution at the tracepoint location, reports a message that execution has reached the tracepoint location, and resumes execution from the point of suspension.

Format

```
SET TRACE[/qualifier] [address-expression]
```

Command Parameters

address-expression

Denotes the location at which the tracepoint is to be established.

Command Qualifiers

/BRANCH

Specifies that tracepoints be established at each of the following members of the family of BRANCH instructions: BNEQ, BEQL, BGTR, BLEQ, BGEQ, BLSS, BGTRU, BLEQU, BVC, BVS, BGEQU, BLSSU, BRB, BRW, JMP, BBS, BBC, BBSS, BBCS, BBSC, BBCC, BBSSI, BBCCI, BLBS, BLBC, ACBB, ACBW, ACBL, ACBF, ACBD, ACBG, ACBH, AOBLEQ, AOBLSS, SOBGEQ, SOBGTR, CASEB, CASEW, CASEL.

/CALL

Specifies that tracepoints be established at each of the following members of the family of CALL instructions: CALLS, CALLG, BSBW, BSBB, JSB, RSB, RET.

Examples

1. DBG>SET TRACE %LABEL 5

2. DBG>SE TR/BRA

3. DBG>SE TR/CA

SET TYPE

Description

Establishes the default type to be associated with untyped program locations and, when the /OVERRIDE qualifier is specified, the type to be associated with both untyped program locations and program locations that have compiler-generated types.

The following debugger types are acceptable parameters in the SET TYPE or SET TYPE/OVERRIDE commands: BYTE, WORD, LONG, QUADWORD, OCTAWORD, FLOAT, D_FLOAT, G_FLOAT, H_FLOAT, ASCII:n, and INSTRUCTION.

Debugger default and override types influence the interpretation and display of program entities identified by address expressions in debugger commands such as EXAMINE, DEPOSIT, and EVALUATE.

Types specified as command qualifiers in debugger commands have precedence over types specified as parameters in the SET TYPE and SET TYPE/OVERRIDE commands.

Format

SET TYPE[/qualifier] type-keyword

Command Parameters

type-keyword

The type-keyword may be any of the following debugger types:

- BYTE -- Type byte integer (length 1 byte).
- WORD -- Type word integer (length 2 bytes).
- LONG -- Type longword integer (length 4 bytes).
- QUADWORD -- Type quadword integer (length 8 bytes).
- OCTAWORD -- Type octaword integer (length 16 bytes).
- FLOAT -- F_floating type (length 4 bytes). Values of type F_floating may range from .29*10**-38 to 1.7*10**38 with approximately 7 decimal digits precision.
- D_FLOAT -- D_floating type (length 8 bytes). Values of type D_floating may range from .29*10**-38 to 1.7*10**38 with approximately 16 decimal digits precision.
- G_FLOAT -- G_floating type (length 8 bytes). Values of type G_floating may range from .56*10**-308 to .9*10**308 with approximately 15 decimal digits precision.
- H_FLOAT -- H_floating type (length 16 bytes). Values of type H_floating may range from .84*10**-4932 to .59*10**4932 with approximately 33 decimal digits precision.

- ASCII:n -- Type ASCII character (length n bytes), where each character occupies one byte of memory. If you do not specify a value for n, the debugger assumes a default length of 4 bytes. The value n is interpreted in decimal radix.

- INSTRUCTION -- Type VAX-11 instruction whose length is variable, depending on the number of instruction operands and the kind of addressing modes used.

Command Qualifiers

/OVERRIDE

Indicates that the specified type be associated with untyped program locations and with those locations that have compiler-generated types.

Examples

1. DBG>SET TYP ASC:8

2. DBG>SET TYP/OVERR LONG

3. DBG>SET TYP D_FLOAT

SET WATCH

Description

Establishes a watchpoint at the location specified by the address expression.

Whenever an instruction causes the modification of a watched location, the debugger takes the following action:

1. Suspends program execution after that instruction has completed execution

2. Reports the watched location

3. Identifies the instruction that modified the watched location (and the line of source code corresponding to that instruction if the SOURCE parameter is in effect by virtue of a previous SET STEP SOURCE command)

4. Reports the value at the watched location before modification

5. Reports the new or modified value at the watched location

6. Issues its prompt

If the watched location has a compiler-generated type, the debugger uses the length in bytes associated with that type to determine the length in bytes of the watched location. If the watched location does not have a compiler-generated type, the debugger watches four bytes of virtual memory beginning at the byte identified by the address expression.

You cannot establish a watchpoint at a dynamically allocated memory location.

Note that if a system service such as one that performs input/output attempts to write to a page of memory that contains one or more watchpoints, either an exception condition is generated or the system service returns failure status; therefore, it is wise to cancel watchpoints before input/output operations are performed.

Format

SET WATCH address-expression

Command Parameters

address-expression

Specifies the location at which the watchpoint is to be established.

Command Qualifiers

None.

Example

```
DBG> SET WAT MAXCOUNT
```

SHOW

Description

Causes the debugger to display the current breakpoints, tracepoints, watchpoints, modes, types, calls, language, log file, modules, output configuration, scope, source, and step conditions when the corresponding keyword is specified.

See the individual command descriptions following for more information.

Format

SHOW keyword[/qualifier] [parameter]

Command Parameters

keyword

Specifies the item to be displayed. Keyword can be BREAK, CALLS, LANGUAGE, LOG, MARGIN, MAX_SOURCE_FILES, MODE, MODULE, OUTPUT, SCOPE, SEARCH, SOURCE, STEP, TRACE, TYPE, and WATCH.

parameter

Depends on the keyword specified.

Command Qualifiers

Depends on the keyword specified.

SHOW BREAK

Description

Causes the debugger to display breakpoints established by the SET BREAK command.

If you established a breakpoint using the /AFTER:n command qualifier with the SET BREAK command, the SHOW BREAK command displays the current value of the decimal integer n, that is, the originally specified integer value minus one for each time the breakpoint location was reached. (The debugger decrements n each time the breakpoint location is reached until the value of n is zero, at which time the debugger takes break action.)

Format

SHOW BREAK

Command Parameters

None.

Command Qualifiers

None.

Example

```
DBG> SHOW BREAK
breakpoint at SUB1\LOOP
breakpoint at MAIN\MAIN+1F DO (EX SUB1\D ; EX/SYMBOL PSL; GO)
routine breakpoint /after:2 at SUB2\SUB2
```

SHOW CALLS

Description

Causes the debugger to display information about the sequence of currently active procedure calls, or the number of call frames on the stack.

The optional parameter n specifies the call count, or the number of call frames to be displayed, by a decimal integer in the range 0 through 32767. If you do not specify the parameter n, information about all call frames is displayed.

For each call frame, the debugger displays one line of information. The first line displayed contains information about the top call frame (the one representing the most recently called procedure); the next line contains information on the next most recently called procedure; and so on.

Each line of information displayed by the debugger contains the name of the called routine, the name of the module that contains the called routine, the line number of the call (in line-oriented languages only), and the value of the PC in the calling routine at the time that control was transferred to the called routine.

The value of the PC is displayed in two ways: as an absolute virtual address and as a virtual address relative to the virtual address of the name of the routine.

Format

```
SHOW CALLS [n]
```

Command Parameters

n

Specifies the number of call frames about which you want information. If omitted, the debugger displays information about all call frames.

Command Qualifiers

None.

Example

```
DBG> SHOW CALLS

module name      routine name      line      rel PC      abs PC

SUB2             SUB2                        00000002    0000085A
SUB1             SUB1                 5      00000014    00000854
MAIN             MAIN                10      0000002C    0000082C
```

SHOW LANGUAGE

Description

Causes the debugger to display the current language.

The current language is the language last established by the SET LANGUAGE command or the language established at debugger start up.

Format

SHOW LANGUAGE

Command Parameters

None.

Command Qualifiers

None.

Example

```
DBG> SH LANG
language: BASIC
```

SHOW LOG

Description

Causes the debugger to display the name of the current log file and to report whether the debugger is writing to that log file.

The current log file is the log file last established by a SET LOG command or the default log file DEBUG.LOG.

Format

SHOW LOG

Command Parameters

None.

Command Qualifiers

None.

Example

```
DBG> SH LOG
not logging to DEBUG.LOG
```

SHOW MARGIN

Description

Displays the current source-line margin settings for the display of source code.

Margin settings are established by the SET MARGIN command. By default, the debugger sets the left margin to 1 and the right margin to 255.

Format

SHOW MARGIN

Command Parameters

None.

Command Qualifiers

None.

Examples

1. ```
   DBG>SHOW MARGIN
   left margin: 1, right margin: 255
   ```

2. ```
   DBG>SET MARGIN 50
   DBG>SHOW MARGIN
   left margin: 1 , right margin: 50
   ```

3. ```
   DBG>SET MARGIN 10:60
   DBG>SHOW MARGIN
   left margin: 10 , right margin: 60
   ```
   ```

SHOW MAX_SOURCE_FILES

Description

Displays the maximum number of source files that the debugger may keep open at any one time.

The maximum number of source files that the debugger may keep open at any one time may be specified using the SET MAX_SOURCE_FILES command or may be the default value of 5.

Format

```
SHOW MAX_SOURCE_FILES
```

Command Parameters

None.

Command Qualifiers

None.

Example

```
DBG> SHOW MAX_SOURCE_FILES
max_source_files: 5
```

SHOW MODE

Description

Causes the debugger to display the current radix mode and the current symbolic or nonsymbolic mode.

The current modes are the modes last established by the SET MODE command or the default modes associated with the current language.

Format

SHOW MODE

Command Parameters

None.

Command Qualifiers

None.

Example

```
DBG> SH MOD
modes: symbolic, decimal
```

SHOW MODULE

Description

Causes the debugger to display the following information about each module in your program:

- The module name
- The language in which the module is written
- Whether or not the Run-Time Symbol Table (RST) contains information about the symbols in that module
- The space (in bytes) required in the RST for symbols in that module
- The total number of modules in the program
- The number of unused bytes in the space allocated for the RST

Format

SHOW MODULE

Command Parameters

None.

Command Qualifier

None.

Example

```
DBG> SHOW MODU
module name        symbols    language    size
FOO                yes        MACRO       432
MAIN               no         FORTRAN     280
SUB1               no         FORTRAN     164
SUB2               no         FORTRAN     204
total modules:  4.            remaining size:  60720.
```

SHOW OUTPUT

Description

Causes the debugger to display the current output configuration.

The current output configuration reflects whether or not the debugger is displaying output on the terminal, whether or not the debugger is writing output to a log file, and whether or not the debugger displays input command strings when it executes command procedures and DO command sequences.

Format

SHOW OUTPUT

Command Parameters

None.

Command Qualifiers

None.

Example

```
DBG> SH OUT
output: noverify, terminal, not logging to DEBUG.LOG
```

SHOW SCOPE

Description

Causes the debugger to display the current scope search list, that is, the scope search list established by the last SET SCOPE command.

The current scope search list designates one or more program locations (specified by pathnames and/or other special characters) to be used in the interpretation of symbols that are specified without pathname prefixes in debugger commands.

Note that if you have used a decimal integer in the SET SCOPE command to represent a called routine, the debugger attempts to display the name of the routine represented by the decimal integer when it executes the SHOW SCOPE command.

Format

SHOW SCOPE

Command Parameters

None.

Command Qualifiers

None.

Example

```
DBG> SH SCO
scope: MAIN, SUB1, 0[=SUB2],\
```

SHOW SEARCH

Description

Displays the current SEARCH parameters.

Current SEARCH parameters are either established by the SET SEARCH command or are the default values NEXT and STRING.

SEARCH parameters determine whether the debugger searches for all occurrences (ALL) of the string or only the next occurrence (NEXT) of the string, and whether the debugger displays any occurrence of the string (STRING) or only those occurrences in which the string is not bounded on either side by a character that can be part of an identifier in the current language (IDENTIFIER).

Format

SHOW SEARCH

Command Parameters

None.

Command Qualifiers

None.

Example

```
DBG>SHOW SEARCH
search settings: search for next occurence, as a string
DBG>SET SEARCH IDENT
DBG>SHOW SEARCH
search settings: search for next occurence, as an identifier
DBG>SET SE AL
DBG>SH SEA
search settings: search for all occurences, as an identifier
```

SHOW SOURCE

Description

Displays the source directory search list(s) currently in effect.

The SET SOURCE/MODULE=modname command establishes a source directory search list for a particular module. The SET SOURCE command establishes a source directory search list for all modules not explicitly mentioned in a SET SOURCE/MODULE=modname command.

If a directory search list has not been established by means of the SET SOURCE or SET SOURCE/MODULE=modname commands, the SHOW SOURCE command indicates that no directory search list is currently in effect. In this case, the debugger expects each source file to be in the same directory as it was in at compile time.

Format

SHOW SOURCE

Command Parameters

None.

Command Qualifiers

None.

Example

```
DBG> SHOW SOURCE
no directory search list in effect
DBG> SET SOURCE [MARY],[JOHN]
DBG> SHOW SOURCE
source directory search list for all modules:
        [MARY]
        [JOHN]
```

SHOW STEP

Description

Causes the debugger to display the current step conditions.

The current step conditions reflect whether the debugger steps by lines or by instructions, whether the debugger steps "into" routines in the user program or "over" them, whether the debugger steps "into" routines in system space or "over" them, and whether the debugger displays lines of source code as it steps.

Current step conditions are the step conditions established by the last SET STEP command or the default step conditions established by the current language.

Format

SHOW STEP

Command Parameters

None.

Command Qualifiers

None.

Example

```
DBG>SH STEP
step type: nosystem, by instruction, over routine calls, nosource
```

SHOW TRACE

Description

Causes the debugger to display tracepoints established by the SET TRACE command.

If the /BRANCH or /CALL command qualifiers were used with the SET TRACE command, the SHOW TRACE command causes the debugger to report that members of either of these families of instructions are currently being traced.

Format

SHOW TRACE

Command Parameters

None.

Command Qualifiers

None.

Example

```
DBG>   SHOW TRACE
tracepoint at CALC\MULT
tracing /CALL instructions: CALLS, CALLG, BSBW, BSBB, JSB, RSB
and RET
```

SHOW TYPE

Description

Causes the debugger to display the current default type or, if the /OVERRIDE command qualifier is specified, the current override type.

Format

SHOW TYPE[/qualifier]

Command Parameters

None.

Command Qualifiers

/OVERRIDE

Causes the debugger to display the current override type.

Example

```
DBG> SH TY
type: long integer
```

SHOW WATCH

Description

Causes the debugger to display the locations at which watchpoints have been established by the SET WATCH command and to report the length in bytes of each watched location.

Format

SHOW WATCH

Command Parameters

None.

Command Qualifiers

None.

Example

```
DBG> SH WATCH
watchpoint at MAIN\ALPHA for 4. bytes
watchpoint at SUB2\TABLE+20 for 4. bytes
```

STEP

Description

Causes the debugger to execute your program by line or by instruction, depending on current STEP conditions.

When you issue a STEP command, the debugger takes the following action:

1. Reports the next instruction or line to be executed
2. Executes an instruction or a set of instructions
3. Reports the instruction or line that follows the last instruction executed
4. Reports the source line corresponding to the line or instruction that follows the last instruction executed only if the SOURCE parameter is in effect by virtue of STEP/SOURCE or SET STEP SOURCE and source lines are available
5. Issues its prompt

If you specify the decimal integer n as a parameter in the STEP command, the debugger executes n instructions or lines.

Each language establishes default STEP conditions. When you change the current language, STEP conditions are set to those specified in that language. You can change these STEP conditions by issuing the SET STEP command.

You can override current STEP conditions by specifying a STEP condition as a command qualifier in the STEP command.

Format

```
STEP[/qualifier...] [n]
```

Command Parameters

n

Specifies the number of lines or instructions to be executed by the STEP command. If you do not specify the parameter n, the debugger executes one line or one instruction.

Command Qualifiers

/INSTRUCTION

A single step causes execution of a single instruction at the location where execution was last suspended.

/LINE

A single step causes execution of all the user code between the location where execution was last suspended and the next line boundary.

/INTO

Steps "into" called routines in the user program space (and "into" called routines in system space if /SYSTEM is also specified). In its execution of steps, the debugger does not differentiate between code within a called routine and code outside of a called routine.

/OVER

Steps "over" called routines both in the user program space and in system space. The debugger considers any code executed as the result of a CALL instruction, up to and including the corresponding RETURN instruction, to be part of a single STEP.

/SYSTEM

Steps "into" called routines in system space, provided that /INTO is also specified. In its execution of steps, the debugger does not differentiate between code within a called routine in system space and code outside a called routine in system space. Thus, execution of a STEP command may result in suspension of execution in system space.

/NOSYSTEM

Steps "over" called routines in system space. The debugger considers any code executed as the result of a CALL instruction, up to and including the corresponding RETURN instruction, to be part of a single STEP.

/SOURCE

Displays the source line corresponding to the instruction or line that follows the last instruction executed as a result of the STEP.

/NOSOURCE

Specifies that a source line not be displayed as a result of this STEP command. STEP/NOSOURCE is used when the SOURCE parameter is in effect by virtue of a previous SET STEP SOURCE command but when source code display is not desired with this particular STEP command.

Examples

1.
```
DBG> STEP
start at MAIN\MAIN+09
stepped to MAIN\MAIN+14: MOVL  222,R0
```

2.
```
DBG> S/LINE
start at MAIN\MAIN+14
stepped to MAIN\MAIN+30: ADDL R0,R1,R3
```

3.
```
DBG> S/INTO
start at MAIN\MAIN+30
stepped to routine SUB1: MOVAL L^0000060C,R11
```

TYPE

Description

Displays the line(s) of source code that correspond to the specified line number(s).

The line numbers used by the debugger to identify lines of source code are generated by the compiler and appear in the compiler listing.

If you specify a single line number, the debugger displays the source code corresponding to that line number.

If you specify a list of line numbers, separating each with a comma (,), the debugger displays the source code corresponding to each of the line numbers.

If you specify a range of line numbers, separating the starting and ending line numbers in the range with a colon (:), the debugger displays the source code corresponding to that range of line numbers.

You can read through all the source language statements in your program by specifying a range of line numbers starting from 1 and ending at a number equal to or greater than the largest line number in the program listing.

After displaying a single line of source code, you can display the next line by issuing a TYPE command without a line number, that is, by issuing a TYPE command and then pressing the RETURN key. You can then display the next line and successive lines by repeating this sequence, in effect, reading through your source program one line at a time.

You can specify a module name with the line number(s) to indicate that the line(s) are located in that module. In this case, you enter the module name, a backslash (\), and the line number(s), without intervening spaces.

If you do not specify a module name with the line number(s), the debugger uses the current scope setting to determine which module to use. In this case, the current scope is either the first module designated in a SET SCOPE command or, if a SET SCOPE command was not issued, the module containing the current PC.

Format

```
TYPE [ [modname\]line-number[:line-number] -
     [,[modname\]line-number[:line-number]...] ]
```

Command Parameters

line-number

Any number generated by a compiler to label a source language statement or statements.

Command Qualifiers

None.

Examples

1.
```
DBG> TYPE 160
module COBOLTEST
   160: START-IT-PARA.
DBG> TYPE (RET)
module COBOLTEST
   161:          MOVE SC1 TO ES0.
```

2.
```
DBG> T 160:163
module COBOLTEST
   160: START-IT-PARA.
   161:          MOVE SC1 TO ES0.
   162:          DISPLAY ES0.
   163:          MOVE SC1 TO ES1.
```

3.
```
DBG> TYPE COBOLTEST\160,22:24
module COBOLTEST
   160: START-IT-PARA.
module COBOLTEST
    22: 02       SC2V2   PIC S99V99       COMP VALUE  22.33.
    23: 02       SC2V2N  PIC S99V99       COMP VALUE -22.33.
    24: 02       CPP2    PIC PP99         COMP VALUE  0.0012.
```

INDEX

INDEX

INDEX

VAX-11
Symbolic Debugger
Reference Manual
AA-D026D-TE

READER'S COMMENTS

NOTE: This form is for document comments only. DIGITAL will use comments submitted on this form at the company's discretion. If you require a written reply and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Did you find this manual understandable, usable, and well organized? Please make suggestions for improvement.

Did you find errors in this manual? If so, specify the error and the page number.

Please indicate the type of user/reader that you most nearly represent.

- ☐ Assembly language programmer
- ☐ Higher-level language programmer
- ☐ Occasional programmer (experienced)
- ☐ User with little programming experience
- ☐ Student programmer
- ☐ Other (please specify) ____________

Name ____________ Date ____________

Organization ____________

Street ____________

City ____________ State ____________ Zip Code or Country ____________

— — Do Not Tear - Fold Here and Tape —

digital

No Postage
Necessary
if Mailed in the
United States

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO.33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

BSSG PUBLICATIONS ZK1-3/J35
DIGITAL EQUIPMENT CORPORATION
110 SPIT BROOK ROAD
NASHUA, NEW HAMPSHIRE 03061

— — Do Not Tear - Fold Here —

Cut Along Dotted Line